Facultad de Informática

¿Está Ethereum capacitado para cubrir las expectativas que se esperan de la tecnología blockchain?

UNIVERSIDAD COMPLUTENSE DE MADRID
CURSO ACADÉMICO 2018/2019

Trabajo de fin de grado
Grado en Ingeniería Informática

Departamento de Sistemas Informáticos y Computación

Autores:

HERMENEGILDO GARCÍA NAVARRO

Director:

JESÚS CORREAS FERNÁNDEZ
GUILLERMO ROMÁN DÍEZ

Agradecimientos

En primer lugar, quiero darle las gracias al Dr. Jesús Correas Fernández, director del proyecto, por depositar su confianza en mí para llevar a cabo este proyecto y porque su interés en ayudar a sus alumnos ha hecho posible que este proyecto haya llegado a su fín y que trabajar en él haya sido un placer.

Dar las gracias a mis padres, por ser un ejemplo para mí en un sinfín de ocasiones y estar dispuestos a ayudarme siempre que lo necesito. A Clara, a Gonzalo y a Paula, por ser los mejores amigos que una persona puede desear y porque vuestra forma de ser llena de luz todo lo que os rodea. Y, para terminar, gracias a mí segunda estrella a la derecha, Marina, gracias por estar a mi lado, por ser tan fuerte y cariñosa y por darme fuerzas cuando más las necesito, gracias porque a tu lado todo lo malo no es tan grave y todo lo bueno es mejor.

Resumen

Sin duda el blockchain ha sido uno de los inventos más revolucionarios de esta década. Desde grandes empresas hasta equipos de investigación dedican sus esfuerzos a estudiar esta tecnología e investigar nuevas formas de utilizarla y mejorarla. Desde el punto de vista de la ingeniería informática tiene especial interés un blockchain en particular, el de Ethereum.

En un momento en el que las criptodivisas se encontraban en auge y parecía que la utilidad del blockchain se limitaba a la creación de estas, Vitalik Buterin y su equipo presentaron Ethereum, un blockchain que mantiene una máquina virtual de propósito general.

Las expectativas de la sociedad sobre este blockchain eran altísimas y se esperaban hazañas como la de ser el nuevo internet. Esta situación de euforia colectiva distorsiona la realidad y hace que parezca imposible realizar un proyecto sobre Ethereum sin que este tenga que suponer un gran cambio. Sin embargo, como Vitalik Buterin comentó en Deconomy, "la tecnología blockchain por sí sola es un ordenador y una base de datos mucho menos eficiente que la tecnología que existía hace cuarenta años" [54].

El objetivo de este proyecto es determinar qué se puede esperar de Ethereum actualmente y en qué debe mejorar antes de poder cubrir las expectativas tan altas que se tienen de este blockchain. Este objetivo se conseguirá a través del análisis de la tecnología blockchain y de las técnicas que permiten que dentro del blockchain de Ethereum haya una máquina virtual.

Abstract

Without any doubt in this decade the blockchain technology is one of the most revolutionary inventions of this decade. Big companies and research teams are investing their efforts in studying this technology and looking for new ways in which it can be used or upgraded. From the computer scientist point of view, the Ethereum blockchain is of special interest.

When cryptocurrencies appeared it looked like that all a blockchain could do was that, but it changed when Vitalik Buterin and his team showed the world Ethereum, a blockchain that embeds a general purpose virtual machine.

Society's expectations on what these blockchain could achieve were really high: Ethereum was expected to be the next internet. This social euforia situation distorts reality and making something inside Ethereum that does not look like a huge change seems impossible. Nevertheless, as was said by Vitalik Buterin in Deconomy, “blockchains by themselves are a far less efficient computer and database than technology that has existed for over 40 years ” [54].

The main goal of this project is to find what can be expected of Ethereum nowadays and what needs to be upgraded in order to fulfill society’s expectations. And it will be achieved by the analysis of blockchain technology and the techniques used by Ethereum that make possible for a virtual machine to be inside a blockchain.

Palabras clave

Blockchain, proof of work, consenso, double spending, Ethereum, máquina virtual,gas, reentrada, vulnerabilidad y verificación formal.

Key words

Blockchain, proof of work, consensus, double spending, Ethereum, virtual machine,gas, re-entrant, vulnerability and formal verification.

Introducción

La tecnología blockchain ha venido para quedarse. Desde el gran éxito de Bitcoin, el mundo esta pendiente de su evolución y aplicaciones. La repercusión ha sido tal que actualmente existen 2202 criptodivisas distintas, muchas de ellas con valor en bolsa [38].

Los blockchains son capaces de almacenar información y de registrar los cambios que en ella se producen, todo ello de forma descentralizada. La utilidad de un blockchain esta ligada al tipo de información que almacena y los cambios que puede registrar. Un blockchain servirá como libro mayor si almacena la cantidad de dinero que posee cada uno de sus usuarios y registra las transacciones entre ellos. Del mismo modo, un blockchain servirá como registro de la propiedad si almacena las propiedades de sus usuarios y registra los cambios de propietario que sufren.

La gran popularidad que alcanzó esta tecnología pronto sacó a la luz un problema, los blockchains no facilitaban implementar nuevas funcionalidades sobre ellos. Si existía un blockchain que servía como libro mayor, hacer que también se usase como registro de la propiedad era demasiado complicado.

Esta situación llevó a Vitalik Buterin a diseñar Ethereum, un blockchain de propósito general. A diferencia del resto de blockchains, Ethereum está diseñado para que cualquiera, a través de un lenguaje de programación, pueda implementar nuevas funcionalidades sobre él. Ethereum consigue esto gracias a que es un blockchain que almacena una máquina virtual y registra los cambios de estado de los programas que en ella se ejecutan.

El hecho de que el blockchain de Ethereum mantenga una máquina virtual, capaz de ejecutar código, lo hace muy interesante desde el punto de vista de la ingeniería informática. Este documento, aunque desarrollará el concepto de blockchain, se centrará en Ethereum y el entorno de ejecución de código que presenta.

Los programas que puede ejecutar la máquina virtual de Ethereum se denominan *smart contracts* y pueden hacer lo mismo que un programa escrito en C++ o Java. Además cualquiera con una cuenta de Ethereum puede, en todo momento, ejecutar un smart contract almacenado en el blockchain de ethereum.

El hecho de que un smart contract se ejecute en un blockchain implica que su ejecución no se puede manipular, es decir, un agente externo no podrá alterar ni la memoria que gestiona ni su código.

Además, Ethereum mantiene una criptodivisa, el *ether*, con un valor actual de $243.18 [38]. En Ethereum todo el que posea una cuenta puede almacenar, transferir y recibir *ether*, incluidos los smart contracts.

A raíz de la aparición del blockchain de Ethereum se definió el concepto de aplicación descentralizada como aquella aplicación cuya lógica esta implementada en smart contracts.

El siguiente smart contract, *Domain*, muestra el *"hello world"* de la programación de smart contracts. *Domain* se encarga de registrar dominios y sus propietarios. Si un usuario quiere registrar un dominio llama a la función *registrar* (línea 7). Esta comprueba si el dominio esta ya en uso y, de no ser así, lo crea y se asigna como propietario a quien envió el mensaje.

Por otro lado la función *eliminar* recibe un dominio, comprueba si esta registrado y, de ser así, lo elimina.

```
1   pragma solidity 0.4.17;
2   contract Domain
3   {
4         mapping (string => bool) existe;
5         mapping (string => address) propietario;
6
7         function registrar(string direccion) external
8         {
9               if(existe[direccion])
10              revert();
11              else{
12                    existe[direccion] = true;
13                    propietario[direccion] = msg.sender;
14              }
15        }
16
17        function eliminar(string direccion) external
18        {
19              if(!existe[direccion])
20                    revert();
21              else
22                    existe[direccion] = false;
23         }
24  }
```

Figura 1: smart contract Domain, un "hello world" en Ethereum.

Un modo de implementar un registro descentralizado de dominios y sus propietarios sería construir un blockchain desde cero con dicha finalidad, con todo lo que ello implica. Gracias a Ethereum solo se necesita desarrollar un smart contract de 24 líneas como el de la figura 1.

El boom que tuvieron las criptomonedas cuando apareció Bitcoin se ha repetido con las aplicaciones descentralizadas desde que apareció Ethereum. La gráfica de la imagen 1, proporcionada por [18], confirma esto. En ella se puede observar cómo el número de nuevas aplicaciones descentralizadas que aparecen cada mes, desde Abril de 2016, es muy elevado. Además la gráfica muestra, en los mismos intervalos de tiempo, cómo en Ethereum el número de aplicaciones descentralizadas lleva aumentando desde sus inicios, albergando en Abril de 2019 cerca de 2500 aplicaciones descentralizadas.

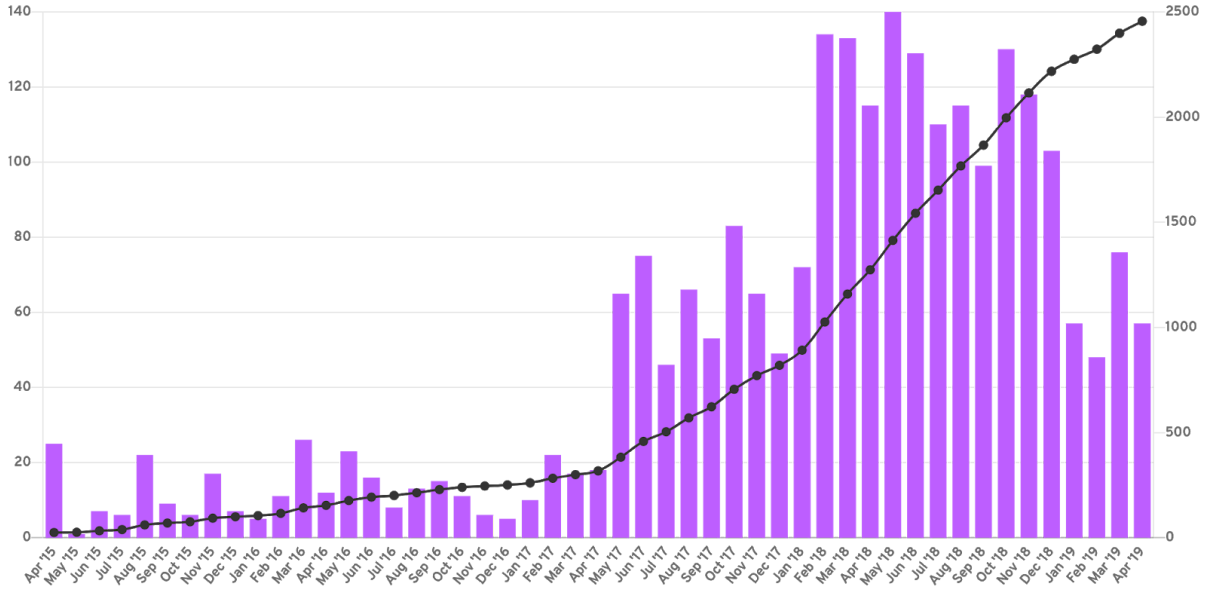


Imagen 1: las columnas relacionan el número de nuevas aplicaciones descentralizadas en Ethereum (números de la izquierda) con un mes determinado. La línea indica el total de aplicaciones descentralizadas que hubo en Ethereum (números de la derecha) un mes determinado.

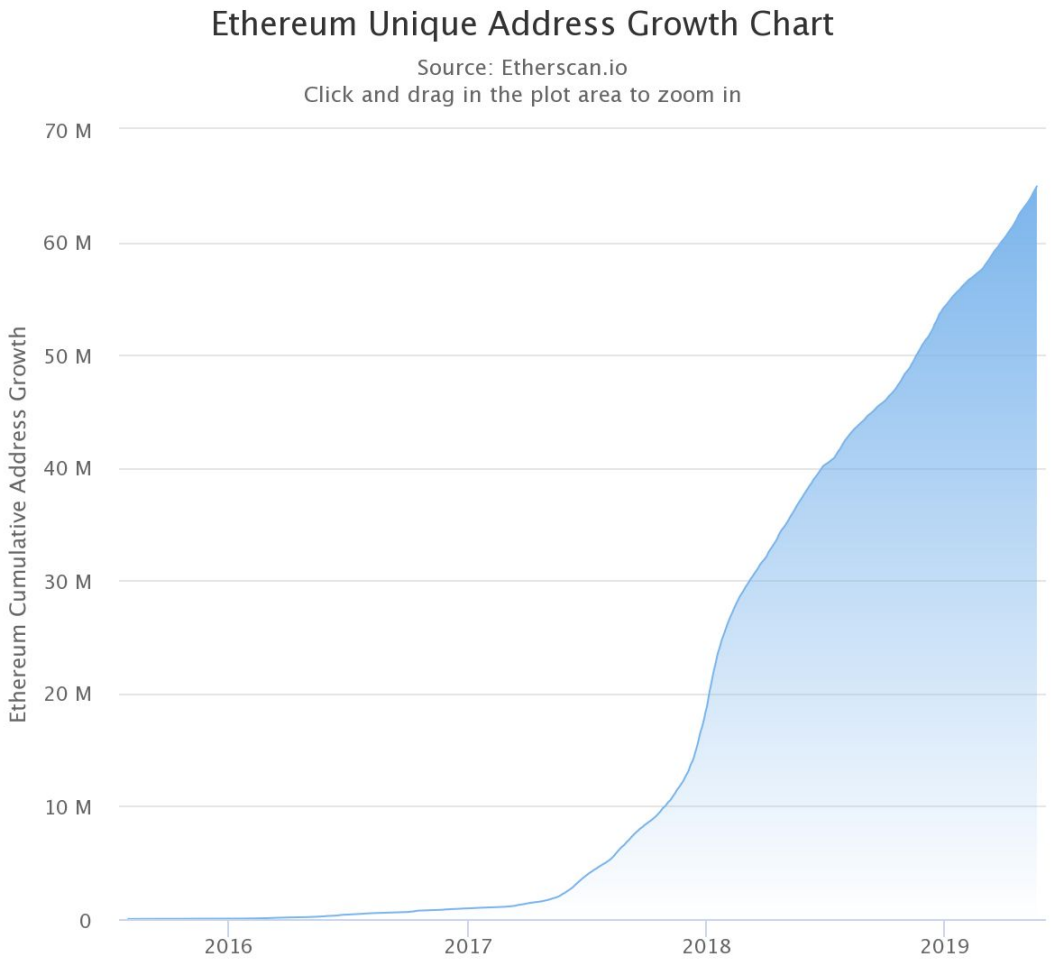


Imagen 2: crecimiento del número de cuentas de Ethereum desde 2016 hasta la actualidad.

Según la web *the state of the dapps* [18], actualmente hay un total de 2.667 aplicaciones descentralizadas en Ethereum, con un total de 3.900 smart contracts y 126.250 usuarios activos.

En la imagen 2 se puede ver cómo el boom de las aplicaciones descentralizadas ha afectado al crecimiento en el número de cuentas de Ethereum desde 2016. Actualmente Ethereum almacena más de 60 millones de cuentas.

¿Está Ethereum capacitado para cubrir las expectativas que se esperan de la tecnología blockchain?

Ethereum presenta un concepto nunca antes visto, una máquina virtual dentro de un blockchain. Es un escenario semejante a lo ocurrido con los *smartphones*, al principio desarrollar aplicaciones para teléfonos móviles, como si de un ordenador se tratase, era impensable. Actualmente los *smartphones* tienen una capacidad de cómputo bastante alta de la que pueden aprovecharse las aplicaciones que en ellos se ejecutan. Los smart contracts son para el blockchain de Ethereum lo mismo que las aplicaciones para los *smartphones*, se encargan de dotar al blockchain de nuevas funcionalidades. Conocer qué aplicaciones se pueden implementar en una máquina virtual situada dentro de un blockchain así como analizar lo idóneo que es Ethereum para implementar dichas aplicaciones son el objetivo de este TFG.

En la primera parte de este TFG se estudiará cómo la tecnología blockchain es capaz de almacenar información y registrar sus cambios de manera descentralizada. Esta capacidad supone un avance tecnológico sorprendente, ya que permite que un conjunto de nodos estén de acuerdo en el estado actual de la información que comparten sin la necesidad de confiar entre ellos o confiar en un tercero. Sin embargo gestionar información a través de un blockchain tiene un coste tanto en tiempo como en dinero. Como se expondrá más adelante, a través del concepto de *proof of work*, alterar el estado de un blockchain requiere más energía y más tiempo de la que requiere un entorno centralizado. El análisis de esta tecnología permitirá conocer las limitaciones que heredará la máquina virtual de Ethereum al estar almacenada en un blockchain y, por lo tanto, los aplicaciones que en ella se ejecuten.

Además de analizar la tecnología blockchain, en la primera parte, se analizará el blockchain de Ethereum. Se estudiará cómo Ethereum consigue que una máquina virtual pueda ejecutarse dentro de un blockchain. En este estudio aparecerán conceptos como el de *gas*, que será determinante a la hora de determinar qué aplicaciones tiene sentido implementar en Ethereum y cuáles no.

La primera parte se concluirá con una serie de limitaciones dentro de las cuales se pueden desarrollar aplicaciones de Ethereum. El lector conocerá las expectativas que se pueden tener de la tecnología blockchain y hasta dónde pueden llevar las aplicaciones que se desarrollen para Ethereum.

En la segunda parte de este TFG se desarrolla una aplicación para Ethereum que excede los límites marcados en la primera parte. La finalidad de esta aplicación es comprobar cómo afecta a la viabilidad de la aplicación. En concreto se verá que una aplicación, al exceder estos límites, sufre de grandes problemas de escalabilidad. Se espera que a través de este ejemplo el lector comprenda que las aplicaciones que podrían ser viables en un entorno de aplicaciones centralizado no tienen porqué serlo en uno descentralizado.

Por otro, lado esta segunda parte expondrá el proceso de desarrollar una aplicación para Ethereum. Una vez se ha almacenada la aplicación en Ethereum, solo tendrá sentido si los usuarios son capaces de interactuar con ella. En esta segunda parte se verán una serie de servicios: Infura, Web3.js, Rinkeby y Metamask, que permiten almacenar una aplicación en Ethereum, probarla y comunicarse con ella. Al final de esta segunda parte las Ethereum no se verá como un ecosistema aislado sino como uno con el que se puede interactuar.

En la tercera parte de este TFG se analizará el desarrollo de aplicaciones para Ethereum desde el punto de vista de la programación de smart contracts. Aun manteniendo una aplicación dentro de los límites marcados en la parte 1 del TFG, ¿Cómo de sencillo es programar dicha aplicación?. En esta parte se comprobará si Ethereum está capacitado para cubrir las expectativas que se esperan de la tecnología blockchain o, por el contrario, aún se encuentra en un estado embrionario que no ha terminado de amoldarse al entorno en el que se encuentra. Al fin y al cabo Ethereum es un blockchain que almacena una máquina virtual que ejecuta código. Lo complejo que sea elaborar código para esta máquina virtual determinará cómo de capacitado está Ethereum para cumplir su cometido.

Las conclusiones obtenidas en estas tres partes permitirán dar una respuesta a la pregunta realizada en el título de este TFG, ¿Está Ethereum capacitado para cubrir las expectativas que se esperan de la tecnología blockchain?

Por otro lado, al final de este TFG se encuentran dos apéndices que son :

- A: dos propuestas reales de aplicaciones explicadas. Se deja al lector la valoración de su viabilidad.
- B: ATOMICCALL, una nueva propuesta de operación para la EVM. Su utilidad y definición formal.

Introduction

Blockchain technology has come to stay. Since Bitcoin's huge success the world is watching to see its evolution and applications. The impact in society has been so great that nowadays there are 2202 different cryptocurrencies, much of those with actual value in the market [38].

Blockchains are capable of storing information and recording its changes, all that in a decentralized environment. The usefulness of a blockchain is linked to the type of information it stores and the changes it can record. A blockchain can be used as a ledger if it stores how much money each of the users have and records the money transfers between them. In the same way, a blockchain can be used as a property registry if it stores the properties of its users and records the changes of the properties owners.

The popularity achieved technology uncovered soon a problem: the blockchains did not make easy to implement new features into them. If there was a blockchain being used as a ledger, adding to it the feature of being used as a property registry was too difficult.

This situation led Vitalik Buterin to the design of Ethereum, a general purpose blockchain. Unlike the rest of the blockchains, Ethereum is designed for everyone that, using a programming language, were able to add new features to it. Ethereum is able to do this because it is a blockchain that stores a virtual machine and records the change of state of the programs running on it.

The fact that the Ethereum blockchain has a virtual machine, capable of executing code, is very interesting from the point of view of computer science. This document, although it will develop the blockchain concept, will focus on Ethereum and the code execution environment it presents.

The programs that the Ethereum virtual machine can run are called smart contracts and can do the same thing as a program written in C ++ or Java. Furthermore, anyone with an Ethereum account can, at any time, execute a smart contract stored in the Ethereum blockchain.

The fact that a smart contract is executed in a blockchain implies that its execution can not be manipulated, that is, an external agent will not be able to alter either the memory it manages or its code.

In addition, Ethereum, maintains a cryptocurrency, the *ether*, with a current value of $243.18 [38]. In Ethereum, everyone who owns an account can store, transfer and receive ether, including smart contracts.

As a result of the emergence of the Ethereum blockchain, the concept of decentralized application was defined as an application whose logic is implemented in smart contracts.

The next smart contract, *Domain*, shows the "Hello World" of smart contracts programming. *Domain* is responsible for registering domains and their owners. If a user wants to register a domain they just need to call the *register* function (Line 7). This checks if the domain is already in use and, if not, it creates it and assigns it to the message owner.

On the other hand, the delete function receives a domain, checks if it is registered and, if so, deletes it.

```
1   pragma solidity 0.4.17;
2   contract Domain
3   {
4         mapping (string => bool) exist;
5         mapping (string => address) owner;
6
7         function register(string direccion) external
8         {
9               if(exist[direction])
10              revert();
11              else{
12                    exist[direccion] = true;
13                    owner[direction] = msg.sender;
14              }
15        }
16
17        function delete(string direction) external
18        {
19              if(!exist[direction])
20                    revert();
21              else
22                    exist[direction] = false;
23         }
24  }
```

Figure 1: smart contract Domain, the "Hello World" on Ethereum.

One way to implement a decentralized registry of domains and their owners would be to build a blockchain from scratch for that purpose, with all that that implies. Thanks to Ethereum, you only need to develop a 24-line smart contract like the one in Figure 1.

The boom that the cryptocurrencies had when Bitcoin appeared has been repeated with decentralized applications since Ethereum appeared. The graph of Image 1, provided by [18], confirms this. This graph shows the growing number of new decentralized applications that appear every month, since April 2015, accumulative. In addition, the graph shows, in the same time intervals, how in Ethereum the number of decentralized applications has been increasing notably since its inception, hosting in April 2019 about 2500 decentralized applications.

According to the web of The State of the DApps [18], there are currently a total of 2,667 decentralized applications in Ethereum, with a total of 3,900 smart contracts and 126,250 active users.

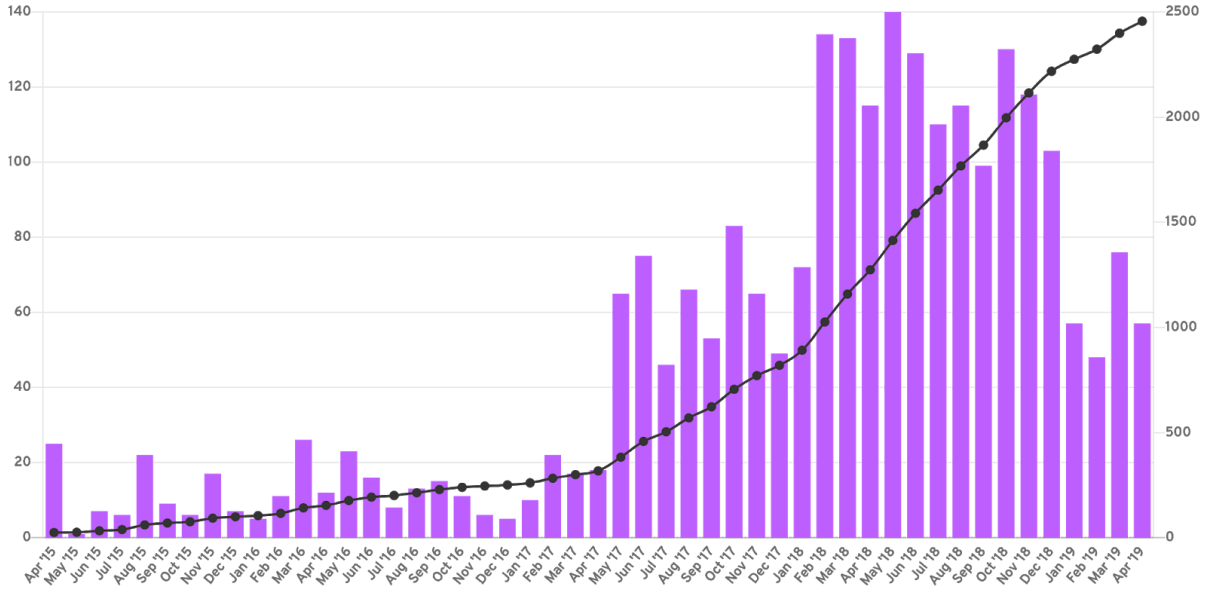


Image 1: the columns relate the number of new decentralized applications in Ethereum (numbers on the left) to a given month. The line indicates the total number of decentralized applications that were in Ethereum (numbers on the right) for a given month.

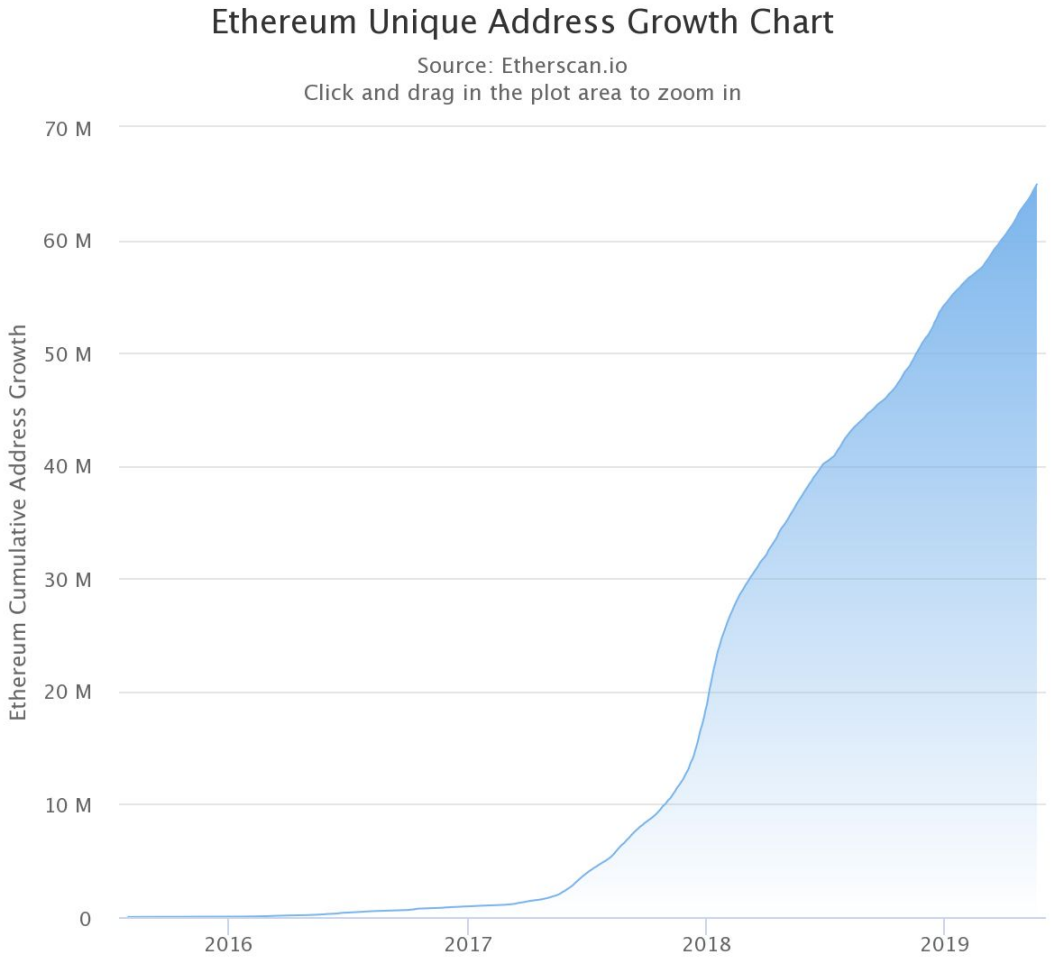


Image 2: growth in the number of Ethereum accounts from 2016 to the present.

The graph in image 2 shows how the boom of decentralized applications has boosted the growth in the number of Ethereum accounts since 2016. Ethereum currently stores more than 60 million accounts.

Is Ethereum capable of meeting the expectations of blockchain technology?

Ethereum shows a concept never seen before, a virtual machine inside a blockchain. It is a scenario similar to what happened with smartphones: at the beginning it was unthinkable to develop applications for mobile phones, just as if they were computers. Currently smartphones have a very high computing capacity can be used by the applications that run on them can be used. The smart contracts are for the blockchain of Ethereum just as the applications for smartphones: they are responsible for providing the blockchain with new functionalities. Knowing what applications can be implemented in a virtual machine located within a blockchain, as well as analyzing what is suitable for executing in Ethereum are the objectives of this work.

In the first part of this TFG we will study how blockchain technology is able to store information and record its changes in a decentralized manner. This capability is a surprising technological advance, since it allows a set of nodes to agree on the current state of the information they share without the need to trust each other or trust a third party. However, managing information through a blockchain has a cost both in time and money. As it will be explained later, through the concept of proof of work, altering the state of a blockchain requires more energy and more time than a centralized environment requires. The analysis of this technology will allow us to know the limitations that the virtual machine of Ethereum will inherit when stored in a blockchain and, therefore, the applications that execute in it.

In addition to analyzing the blockchain technology, in the first part, the blockchain of Ethereum will be analyzed. We will study how Ethereum gets a virtual machine to run inside a blockchain. In this study, concepts such as *gas* will appear, which will be decisive when determining for which applications it makes sense to implement in Ethereum and which ones do not.

The first part will conclude with a series of limitations within which applications of Ethereum can be developed. The reader will know what can we expect from the blockchain technology and how far it can take the applications developed for Ethereum.

The second part of this work develops an application for Ethereum that exceeds the limits set in the first part. The purpose of this application is to check how it affects the viability of the application. Specifically, it will be seen that an application, when exceeding these limits, suffers from several scalability problems. It is hoped that through this example the reader will

understand that the applications that could be viable in a centralized application environment do not have to necessarily be in a decentralized one.

The second part will also show how is the process of developing an application for Ethereum. Once it has been stored in Ethereum, an application only makes sense if users are able to interact with it. We you will see a series of services, Infura, Web3.js, Rinkeby and Metamask, that allow us to store an application in Ethereum, test it and communicate with it. At the end of this second part, Ethereum will not be seen as an isolated ecosystem but as one with which it can interact.

In the third part of this work the development of applications for Ethereum will be analyzed from the point of view of smart contracts programming. Even keeping an application within the limits marked in Part 1, how easy is it to program this application?. In this part we will check if Ethereum is able to meet expectations placed the blockchain technology or, on the contrary, it is still in an embryonic state that has not finished adapting to the environment in which it is located. At the end of the day Ethereum is a blockchain that stores a virtual machine that executes code. How complex it is to develop code for this virtual machine will determine how qualified Ethereum is to fulfill its mission.

Finally, he conclusions obtained in these three parts will make it possible to give an answer to the question asked in the title of this end of degree work. Is Ethereum capable of meeting the expectations placed on of the blockchain technology?

At the end of this document there are two appendices with additional information:

- A: two real proposals of decentralized applications explained. It is left to the reader the assessment of its viability.
- B: ATOMICCALL, a proposal for a new operation for the EVM. Its usefulness and definition formal.

Parte 1

Introducción a la parte 1

Aunque es común escuchar que son el futuro, ante la pregunta "¿Qué puede hacer una aplicación descentralizada?" no es sencillo encontrar una respuesta.

Por otro lado, los casos de uso de aplicaciones descentralizadas son numerosos y muy distintos entre sí. Se pueden encontrar desde juegos en los que se crían mascotas virtuales [42] hasta proyectos tan ambiciosos como la creación de una sociedad sin fronteras gobernada a través de una aplicación descentralizada [41].

El objetivo de esta parte es eliminar todas las luces de neón que rodean el concepto aplicación descentralizada y aportar una respuesta clara a qué son y si son mejores que una aplicación normal. Estas respuestas permitirán vislumbrar los límites de las aplicaciones descentralizadas y aportar un punto de vista realista de sus posibilidades.

Para alcanzar este objetivo se estudiarán los elementos que permiten el desarrollo y funcionamiento de aplicaciones descentralizadas, y se analizará cómo afectan a las limitaciones de estas.

1. Blockchain, el terreno de juego de Ethereum

Qué es, cómo funciona y por que todo el mundo habla de ello

Hace 11 años y bajo el pseudónimo de Satoshi Nakamoto se publicó un artículo [1] en el que se presenta por primera vez el concepto de blockchain y se describe cómo este sistema puede utilizarse para construir una criptomoneda, Bitcoin.

Un año después se hizo disponible al público la primera versión del software de Bitcoin, una criptomoneda cuyo valor actual puede verse en [3].

1.2. ¿Qué es un blockchain?

Un blockchain es una estructura de datos que, gracias al protocolo a través del cual se interactúa con ella y a ciertos conceptos de criptografía, se puede mantener de forma descentralizada sin perder en ningún momento la capacidad de conocer su estado actual.

El mecanismo clásico utilizado para mantener información compartida entre varios individuos es el de recurrir a una entidad central de confianza que guarde dicha información y realice los cambios pertinentes.

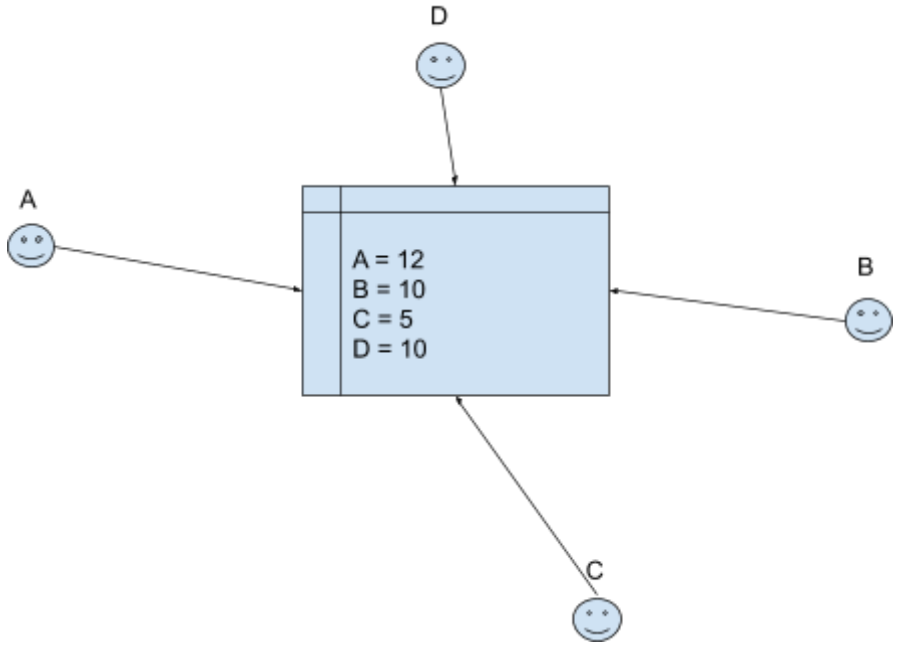


Figura 2: los actores mantienen una única copia del archivo

Para las explicaciones se van a usar cuatro individuos: A, B, C y D. Cada uno de ellos tiene un número de tokens, y quieren gestionar las transacciones de tokens que se hagan entre ellos. Han definido que todas las transacciones tendrán la forma “X paga a Y k tokens”

La figura número 2 muestra, de forma esquemática, un sistema centralizado con los cuatro individuos, A, B, C y D, que comparten un fichero que está en posesión de una entidad

externa de confianza. En este fichero mantienen la información de cuántas unidades monetarias, *tokens*, posee cada individuo. A este fichero se le llamará *fichero de tokens*.

Las figuras 3 y 4 simulan el comportamiento de un usuario y un servidor en un escenario como el de la figura 1. La figura 3 muestra el comportamiento de los usuarios A, B, C y D. Cada usuario puede verse como un proceso que, de manera repetitiva, crea una acción que quiere realizar sobre el fichero de tokens compartido (línea 3), la envía al servidor que posee el fichero para que la realice (línea 4) y después espera a que el servidor le envíe el nuevo estado del fichero de tokens (línea 5). Una vez ha obtenido el nuevo estado, lo imprime.

```
1  Process User[i = 0 to n-1]
2        While(true)
3              A = crea_acción;
4              Enviar_acción_a_realizar(A);
5              E = Recibir_nuevo_estado;
6              print(E);
7        Fwhile
8  Fprocess
```

Figura 3: proceso que simula a un usuario

La figura 4 simula el comportamiento del servidor que posee el fichero que comparten los usuarios A, B, C y D. El servidor es un proceso que, de manera repetitiva, espera a recibir una acción que realizar sobre el fichero de tokens (línea 3), una vez obtiene una acción la ejecuta sobre el fichero (línea 4) y por último envía el fichero de tokens modificado a todos los usuarios (línea 5).

```
1  Process Server
2        While(true)
3              A = Recibir_acción;
4              Cambiar_información(A)
5              Enviar_nuevo_estado;
6        Fwhile
7  Fprocess
```

Figura 4: proceso que simula a un servidor.

Si se quiere eliminar la necesidad de confiar en un servidor que mantenga la información segura y actualizada se debe recurrir a un sistema descentralizado. Para esto lo primero que debe tenerse en cuenta es que cada nodo se encargará de guardar y mantener una copia del fichero de tokens.

La figura 5 representa de forma esquematizada un sistema descentralizado. Ahora los usuarios (A, B, C y D) no cuentan con una entidad centralizada que pueda gestionar los cambios

en el fichero de tokens. Por ello cada usuario posee su propio fichero de tokens y cada vez que alguno realice un cambio lo notificará al resto a través de una red que los conecta a todos.

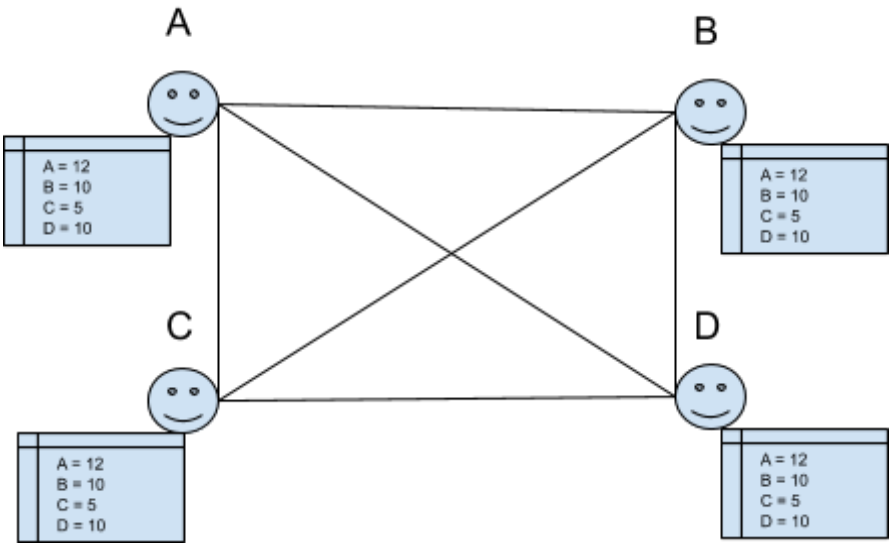


Figura 5: cada nodo mantiene su propia copia del fichero.

La figura 6 muestra el comportamiento de los usuarios en un sistema como el de la figura 5. Un usuario posee su propia copia del fichero de tokens, *I*,. A través de *listen* (línea 3) un usuario recibe las acciones que han realizado el resto de usuarios sobre sus ficheros (línea 5) y las aplica a su fichero de tokens *I* (línea 6) de este modo todos los usuarios coinciden en el estado del fichero. Por otro lado , a través de la función *send* (línea 10), un usuario envía a toda red las acciones que ha realizado sobre su fichero.

```
1   Tipo  User
2         I : Estado;
3         Process listen
4               While(true)
5                     A = Recibir_acción;
6                     Cambiar_información(I,A);
7               Fwhile
8         Fprocess
9
10        function send(acción: A)
11              Emitir_acción(A);
12        Ffunction
13  Ftipo
```

Figura 6: representación del comportamiento de un usuario de la figura 3

El problema de esta solución reside en que los usuarios de la red no tienen por qué comportarse como marca la figura 6. Nada impide a un usuario obviar alguna de las acciones que ha enviado otro usuario del mismo modo que nada le impide comunicar sus acciones a solo

parte de los usuarios. Si un usuario realiza este tipo de acciones provocará que los ficheros de tokens de los usuarios dejen de estar sincronizados y que al cabo de un tiempo sea imposible determinar el estado real del fichero de tokens.

A lo largo de los siguientes apartados se presentarán los distintos problemas que encuentran estos individuos y cómo tras resolverlos llegan a la implementación de un blockchain.

1.3. ¿Quién es quién?

Una de las operaciones que debe registrar este blockchain es el envío de tokens de un nodo a otro. En la figura 7 se muestra cómo D le envía 1 token a A y lo comunica al resto de usuarios de la red, lo cual desenlaza en la situación de la figura 8, en la que todos los nodos han registrado dicha operación.

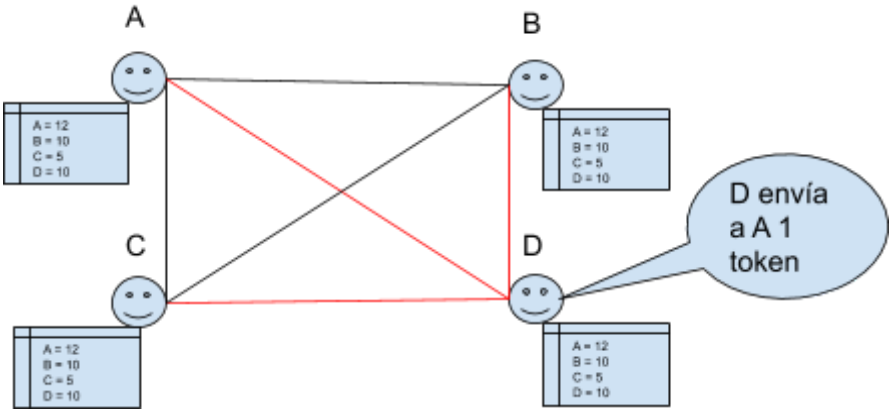


Figura 7: nodo D envía a A 1 token correctamente.

En la figura 8 todos los usuarios de la red han recibido el mensaje de D que expresa su voluntad de enviar 1 token a A y por lo tanto cada uno de ellos lo registra en su fichero de tokens.

Sin embargo el blockchain debe lidiar con la posibilidad de que otro nodo, C por ejemplo, decida hacer trampas y enviar una transacción a la red de la forma "D paga a C 9". El sistema debe asegurarse de que D está de acuerdo con dicha transacción ya que si no C le habría quitado a D todos sus tokens sin permiso de D.

Para que la información del blockchain esté protegida contra este tipo de situaciones se incorpora dentro del sistema el mecanismo ya utilizado en Internet de clave pública/clave privada [4]. Cada nodo del sistema cuenta con una clave pública y una clave privada. De este modo una transacción de la forma "D paga a C 1 token" debe estar firmada por la clave privada de D. Así los demás nodos de la red pueden comprobar que D es quien ha enviado esta transacción descodificando la transacción con la clave pública de D.

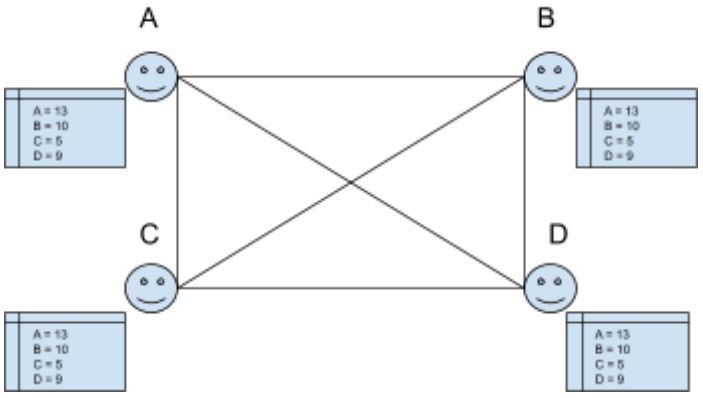


Figura 8: todos los nodos han registrado dicha operación.

```
1   Tipo  User
2         I : Estado;
3         Private_key : key;
4         Public_key : key;
5               N : natural
6         Process listen
7               While(true)
8                     (A,signature) = Recibir_acción;
9                     Sender = A.sender;
10                    If verify(A, signature, Sender.Public_key)
11                          Cambiar_información(I,A);
12                    Fif
13              Fwhile
14        Fprocess
15
16        Function send(acción: A)
17              Signature = Sign(A, this.Private_key, N);
18                          N++;
19              Emitir_acción(A, Signature);
20        Ffunction
21  Ftipo
```

Figura 9: comportamiento de un usuario si se añade un sistema de clave pública - clave privada a la red de la figura 2

Pero esto no es todo, una vez D envía su transacción firmada “D paga a C 1 token”, C podría volver a hacer trampas y copiar esa transacción válida y enviarla tantas veces como quiera. Para solucionar esto, los nodos llevan la cuenta de cuántas transacciones ha enviado cada uno y dentro de cada transacción se incluye el número de transacción que le corresponde. Las transacciones tendrán la forma “1. D paga a C 1 token”, que indica que es la primera transacción de D y que le paga un token a C. Como la firma con la clave privada cambia con el mensaje, si C enviase la transacción “2. D paga a C 1 token” con la misma firma se trataría de una transacción no válida.

En la figura 9 puede verse cómo este cambio en la red afecta al comportamiento de los usuarios descrito en el código de la figura 6. Ahora un usuario deberá almacenar su clave privada y su clave pública (líneas 3 y 4) así como el número de transacciones que ha emitido. Los usuarios ahora reciben y envían una tupla denominada transacción y formada por una acción y su firma (líneas 8 y 17). Cuando un usuario recibe una transacción primero comprueba que la firma es correcta (línea 10) y, de ser así, aplica la acción a su fichero de tokens. Por otro lado, cuando un usuario envía una acción a través de la función *send*, primero se crea la transacción que incluirá la acción y su firma (17), después se incrementará en uno el número de transacciones realizadas, N (línea 18), por último se envía la transacción al resto de usuarios (línea 19).

1.4. La confianza da asco

El sistema de clave pública/clave privada permite conocer si una transacción la ha realizado quien debe. Sin embargo, algo que se debe tener claro a la hora de intentar entender cómo funciona un blockchain público es que **no existe confianza entre los nodos** y el protocolo debe asegurar que todos los nodos están de acuerdo en cuál es el estado actual de la información.

Por ejemplo, cuando B recibe 5 tokens, cómo puede estar seguro de que todos los demás nodos han recibido y apuntado esa transacción. B debe estar seguro de que va a poder usar esos 5 tokens en todo momento.

Para observar cuándo podría ocurrir una situación en la que un nodo recibe tokens pero no puede usarlos, veamos el siguiente ejemplo: en este caso el nodo C, que sólo tiene 5 tokens, se encuentra en una situación en la que le debe 5 tokens a A y otros 5 a B, 10 tokens en total. Como no tiene tantos tokens, C ha diseñado una estrategia que hará que tanto A como B crean que han recibido sus tokens pero C no se haya gastado ninguno.

C, como indica la figura 9, enviará una transacción que solo A puede escuchar y que dice “C paga a A 5 tokens”. A queda contento pues ha recibido sus tokens, sin embargo es el único que lo sabe ya que los demás no han recibido la transacción y por lo tanto A no podrá gastar esos 5 tokens con nadie de la red.

Como sólo A recibe la transacción, su fichero de tokens es el único que registra el cambio y la red queda en una situación en la que hay dos versiones distintas del fichero.

Simultáneamente, como indica la figura 10, C paga a B 5 tokens y por lo tanto emite una transacción que A no puede recibir en la que indica “C paga 5 tokens a B”.

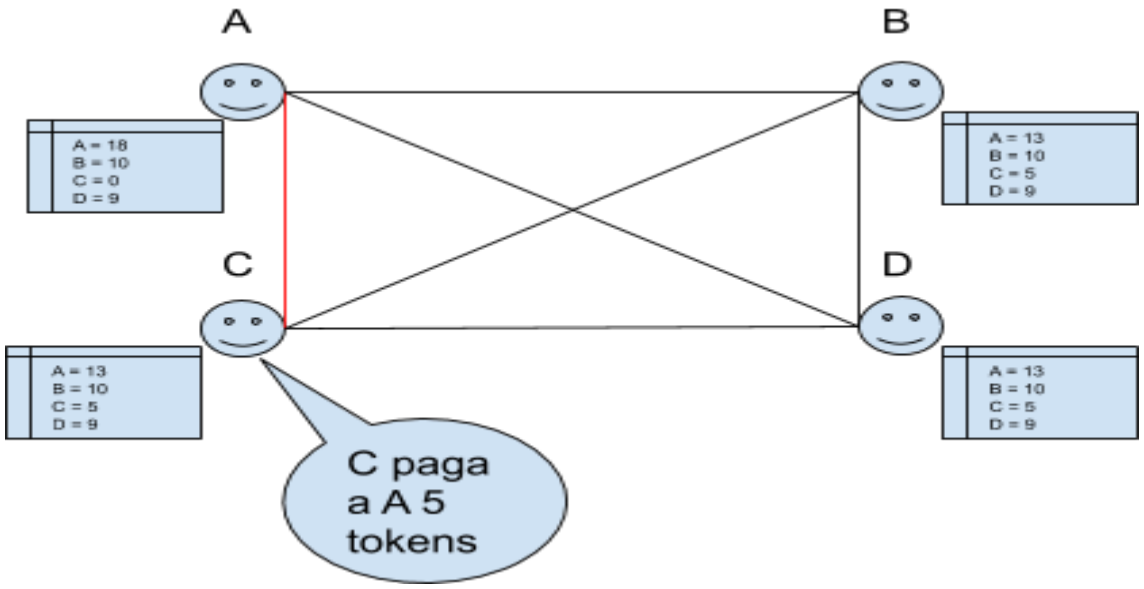


Figura 9: C envía una transacción que solo recibe A.

En la siguiente figura C envía una transacción que sólo B recibe y que indica que le envía 5 tokens. Como sólo B recibe esta transacción su fichero de tokens es el único que registra el cambio, dejando la red con tres versiones del fichero de tokens.

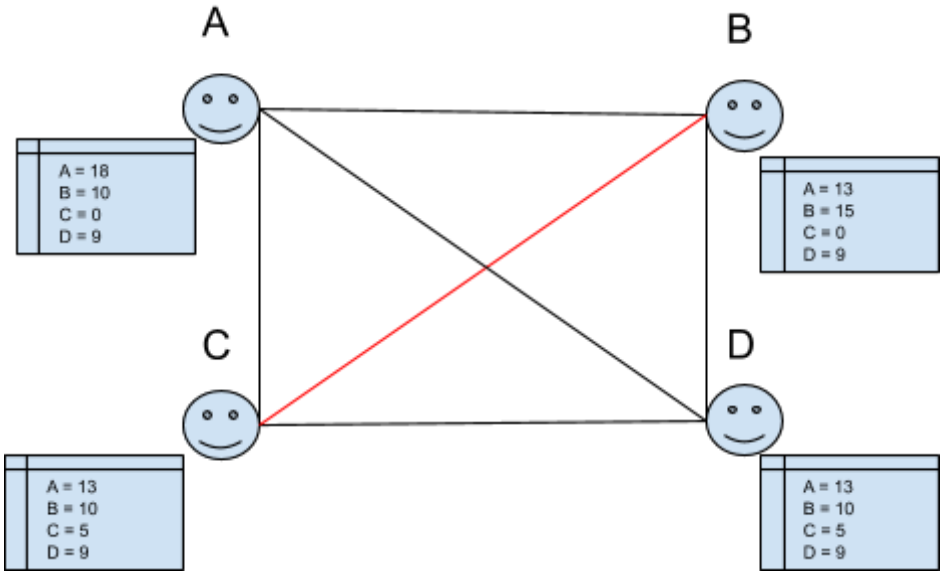


Figura 10: C envía una transacción que solo B puede ver.

B ahora está en la misma situación que A: ha recibido 5 tokens que solo él ha contabilizado. C se ha gastado dos veces los mismos 5 tokens y ha dejado un sistema con distintas versiones acerca del estado de la distribución de tokens. Si todos los nodos intentan ponerse de acuerdo en qué versión es la correcta, no lo conseguirán a no ser que decidan confiar en la versión de alguno de los nodos. Este problema es conocido como **double spending** y el artículo de Satoshi Nakamoto [1] fue el primero en aportar una solución.

1.5. Solucionando el *double spending*, un poco de estructuras de datos

Todos los nodos del sistema deben estar de acuerdo en cuál es el estado actual del fichero de tokens. Para conseguir esto el enfoque que propone el artículo de Satoshi Nakamoto es, en primer lugar, que los nodos guarden no solo el estado actual si no todos los estados por los que ha pasado la distribución de tokens. Para ello se crea una una lista enlazada de bloques siendo un bloque un recipiente que contiene el enlace a su predecesor, las transacciones que se han aplicado al fichero de tokens del bloque predecesor y el estado actual del fichero de tokens.

Además dentro de la red se introduce un nuevo rol, el de minero. Los mineros se encargan de escuchar las transacciones de la red y, cuando han escuchado un número determinado de ellas, crean un nuevo bloque y lo transmiten a la red. Para simplificar este ejemplo, A,B,C y D, además de ser usuarios de la red, serán mineros.

Las figuras 11 y 12 expresan el comportamiento de un minero en la red y cómo estos cambios afectan al comportamiento de los usuarios.

La figura 11 representa el comportamiento de un minero. Un minero posee su copia del blockchain, que es una lista enlazada de bloques (línea 2). El minero crea un nuevo bloque que espera llenar de transacciones. Cuando recibe una transacción primero comprueba si es correcta (línea 9, la firma debe coincidir con la acción enviada y la clave pública) y, de ser así, la incluye en el nuevo bloque. Cuando en este nuevo bloque no se pueden almacenar más transacciones el minero *mina* dicho bloque (proceso explicado en la línea 20) y lo envía a todos los actores (usuarios y mineros) de la red (14). Por último crea un nuevo bloque con el que comienza de nuevo el proceso ya descrito.

Además, para que los nuevos bloques minados sean válidos para toda la red, un minero debe tener su copia del blockchain siempre actualizada. Por este motivo el tipo *Miner* cuenta con el proceso *listen*, donde recibe los nuevos bloques que, de ser válidos (instrucción línea 26 y explicación en línea 32), debe incluirlos en su versión de blockchain para que esté actualizada.

La figura 12 representa el comportamiento de un usuario en la red antes descrita. A los atributos que ya tenía un usuario en la versión anterior (figura 9) se le incluye la copia del blockchain que observa dicho usuario y que debe mantener actualizada(línea 2). Un usuario, al contrario que un minero, no posee un proceso *Mine,* pero sí el proceso *listen*, que es idéntico al utilizado por el tipo *Miner* (figura 11). A través del proceso *listen* un usuario mantiene su blockchain actualizado. Además cuando un usuario recibe una acción (línea 20) se encarga de firmarla, incrementar el número de transacciones que ha emitido y transmitirla a los mineros para que se incluya en un bloque futuro (línea 23).

```
Tipo Miner
      Blockchain : Lista enlazada de bloques;
      Process Mine
            B: bloque;
            B = nuevo_bloque();
            While (true)
                  (A,Signature) = escuchar_acción;
                Sender = A.sender;
                If verify(A, signature, Sender.Public_key,)
                      Añadir_acción(B, A, Signature);
                Fif
                  If bloque_lleno(B)
                      minar_bloque(B);
                      emitir_bloque(B);
                      B = nuevo_bloque();
                  Fif
            Fwhile
      Fprocess

      Function minar_bloque(B) := encuentra el último bloque que fue  minado en el
blockchain y le adhiere el bloque B a continuación.

     Process listen
            While(true)
                  B = Recibir_bloque;
                  If bloque_válido(B)
                      Añadir_bloque(Blockchain, B)
                  Fif
            Fwhile
      Fprocess

     Function bloque_válido (B) := comprueba que el bloque B está enlazado a algún
bloque del blockchain que las transacciones están firmadas correctamente y que al
aplicarlas sobre su predecesor se llega al fichero de tokens de B.
Ftipo
```

Figura 11: comportamiento de un minero en una red sin double spending.

En la figura 13 puede verse el problema que tiene este protocolo: nada le impide a los mineros hacer trampas y minar bloques que generen bifurcaciones en el blockchain. En esta figura el bloque 2 tenía como continuación solo al bloque 3. Sin embargo, por algún motivo desconocido, el contenido del bloque 3 no le venía bien a alguno de los mineros y por ello ha decidido minar el bloque 3' precedido también por el bloque 2. En este momento, mientras unos mineros están siguiendo la cadena que pasa por el bloque 3, otros están continuando la que pasa por el bloque 3'.

El problema de la situación de la figura 13 es que los usuarios y los mineros deben ponerse de acuerdo en qué rama de la cadena es la correcta, cosa que debería determinar el protocolo.

```
1   Tipo  User
2         Blockchain :  Lista enlazada de bloques;;
3         Private_key : key;
4         Public_key : key;
5         N : natural
6
7         Process listen
8               While(true)
9                     B = Recibir_bloque;
10                    If bloque_válido(B)
11                          Añadir_bloque(Blockchain, B)
12                    Fif
13              Fwhile
14        Fprocess
15
16        Function bloque_válido (B) := comprueba que el bloque B está enlazado a algún
17  bloque del blockchain que las transacciones están firmadas correctamente y que al
18  aplicarlas sobre su predecesor se llega al fichero de tokens de B.
19
20        Function send(acción: A)
21              Signature = Sign(A, this.Private_key, N);
22                          N++;
23              Emitir_acción(A, Signature);
24        Ffunction
25  Ftipo
```

Figura 12: comportamiento de un usuario en una red sin double spending.

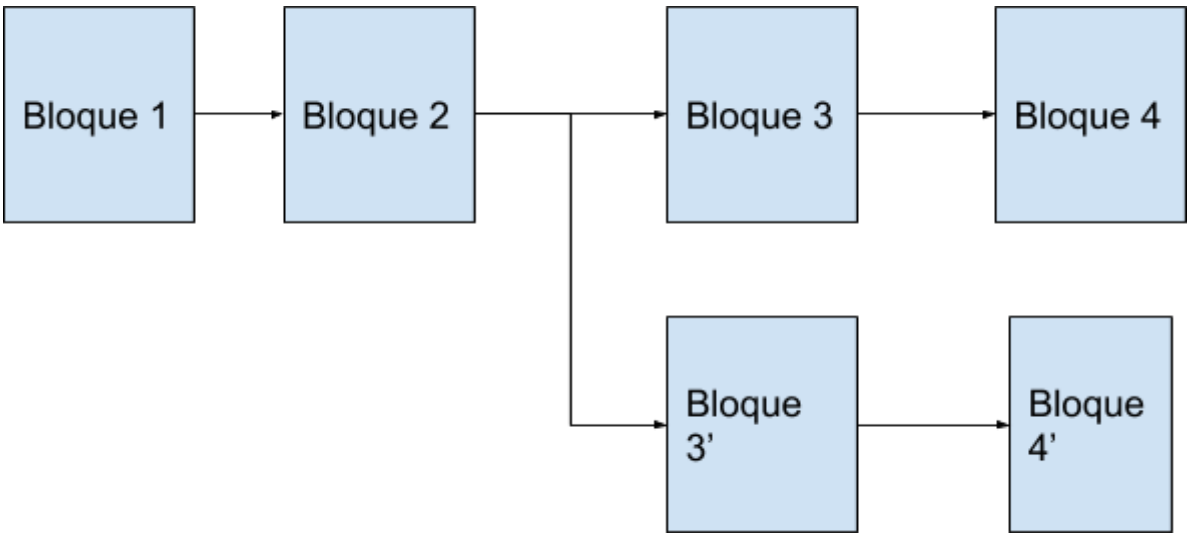


Figura 13: algunos mineros de la red han minado bloques cuyo predecesor ya tenía continuación

La solución propuesta en [1] es que los actores de la red esperen a que en el blockchain haya una rama más larga que las demás y, cuando esto ocurra, esa será la rama correcta. Esta solución necesita que se modifique la forma en la que se mina un bloque ya que, de no ser así, el estado del blockchain podría ser controlado por un grupo de mineros con una capacidad de cómputo mayor a la del resto, ya que minarían bloques más rápido.

1.6. Lo que importa es el trabajo

Es importante introducir algún tipo de mecanismo que transforme minar un bloque en una tarea que cuesta un tiempo más o menos fijo, independientemente de la capacidad de cómputo de los nodos. Minar debe ser una tarea computacionalmente costosa.

En primer lugar, para enlazar los bloques se utilizará la función SHA 256, una función de hash extensamente utilizada en el intercambio de información a través de Internet. El mecanismo de enlazado consiste en incluir dentro de un bloque la salida generada por la función SHA 256 cuando toma como entrada el bloque anterior.

Dentro del bloque se añade un campo de 32 bits llamado **nonce**, el minero se encarga de darle el valor que quiera a este campo. Sin embargo para que un bloque esté minado correctamente se debe cumplir que la función SHA 256, al tomar dicho bloque como entrada, genera un hash con un número determinado de ceros al comienzo. El número de ceros lo determina una variable global del sistema que aumenta y disminuye dependiendo de la velocidad de minado del sistema. Esta información se incluye dentro del bloque en un campo llamado **difficulty** y la probabilidad de encontrar un nonce válido para el bloque es *1/2^difficulty* . Por lo tanto la función *minar_bloque(B)* pasa a actuar como indica la figura 14.

En esta figura se expone cómo funciona el proceso de minar un bloque. Dado el bloque correctamente formado *B* (contiene todos los campos necesarios) la función va cambiando el valor del campo *nonce* en B (línea 4) hasta que la salida producida al introducir *B* en la función SHA 256 tiene tantos ceros al comienzo como indica el campo *difficulty* del bloque.

```
1  Function minar_bloque(B)
2        B.nonce = 0x0;
3        While (número_de_ceros_al_comienzo( SHA-256(B) ) != B.difficulty )
4              Cambiar (B.nonce);
5        Fwhile
6  Ffunction
```

Figura 14: función que representa los pasos a seguir para minar un bloque.

En el caso de Bitcoin, el campo **difficulty** varía para mantener el tiempo de minado de un bloque constante (10 minutos en Bitcoin). Debido a que la función *minar_bloque* es muy costosa y consume mucha energía, quien consigue minar un bloque es recompensado (con 12,5 BTC en Bitcoin).

El proceso definido para minar un bloque se denomina *proof of work* y actualmente es el más utilizado a la hora de implementar un blockchain.

En este entorno, dado que a todos los nodos les cuesta lo mismo minar bloques y que al cambiar uno se deben volver a minar todos los bloques posteriores para que sus hashes coincidan, un nodo sólo podrá controlar el blockchain a su voluntad si posee el 51% de la red, ya que en caso contrario le será imposible minar bloques más rápido que el resto de la red.

En los blockchain públicos actuales, como el de bitcoin, los nodos antes de incluir un bloque en su blockchain realizan una serie de comprobaciones que determinan si el bloque es válido o no, de este modo disminuyen la frecuencia con la que ocurren los forks. Un ejemplo de esto es que si un nodo recibe un bloque cuyo predecesor está muy atrás en el blockchain lo ignora. Además, debido al tiempo que se tarda en minar un bloque, un nuevo bloque contiene un número alto de transacciones nuevas. Así cada nuevo bloque es el resultado de la ejecución de varias transacciones.

En este punto se ha conseguido un sistema descentralizado con un protocolo que aporta la seguridad que daría un sistema centralizado sin tener que depositar la confianza en nadie. De hecho, aunque los nodos fuesen enemigos entre sí, podrían compartir un blockchain estando seguros de que no pueden engañarse entre ellos a través de él.

1.7. Sobre los protocolos de consenso

Del coste total de minar bloques para un blockchain que utiliza proof of work, un 90% es gasto en electricidad [2]. Un precio que los mineros están dispuestos a pagar debido a la recompensa obtenida por minar un bloque.

La principal crítica que se le hace a los blockchain que utilizan proof of work es la energía que se consume en el proceso de minado y el impacto en el medio ambiente que esto trae consigo.

Se estima que el minado en Bitcoin representa el 2% de la energía que se consume en el mundo, lo que equivale al consumo eléctrico de Irlanda de un año [14]. Sin embargo, no es tan importante la cantidad de energía eléctrica que consume Bitcoin sino las emisiones de carbono que genera producir esta electricidad.

Actualmente los mineros sitúan sus centros de minado en zonas donde la energía eléctrica es barata. El sitio tradicional en el que se sitúan las máquinas de minado es China, donde el carbón genera un 60% de la electricidad de la nación y por lo tanto provocando grandes emisiones de carbono.

Una forma de paliar el impacto en el medio ambiente del minado mediante proof of work es situar las máquinas de minado en sitios cuya energía eléctrica se genere a partir de medios con pocas emisiones de carbono (energías renovables). En Europa un sitio popular para el minado de Bitcoin es Islandia, cuya producción de energía eléctrica proviene en su totalidad de energías renovables (energía hidroeléctrica y geotérmica) baratas y con pocas emisiones de carbono [2].

Por otro lado se están implementando otro tipo de protocolos de consenso llamados **proof of stake** capaces de reducir el gasto energético de los blockchain con proof of work. Este tipo de protocolos, aunque prometedores, aún contienen problemas por solucionar que los hacen más vulnerables a la manipulación que los protocolos de proof of work.

2. Ethereum

Blockchain más allá de un libro contable.

Cuando Bitcoin empezó a ganar fama la gente pensó en lo útil que podría llegar a ser la tecnología blockchain. Esta tecnología podía utilizarse no solo para crear criptodivisas sino también para otras cosas.

Durante un tiempo, para conseguir implementar un caso de uso de blockchain había que construir un blockchain que lo soportara o había que implementarlo sobre el blockchain de Bitcoin. La alternativa que solía escogerse era la segunda. Algunos casos de uso que se desarrollaron así fueron los siguientes:

1. *Namecoin*: sirve como una base de datos descentralizada para registrar aquellos identificadores que deben ser únicos [8].
2. *Colored coins*: se trata de un protocolo sobre Bitcoin que permite crear nuevas criptodivisas en el blockchain de Bitcoin [9].

Namecoin fue el primer blockchain que planteó que la tecnología blockchain podía ser utilizada para algo más que para crear criptodivisas. La gente, a raíz de la creación de *Namecoin*, empezó a preguntarse qué cosas pueden hacerse con un blockchain. Las respuestas a esta pregunta eran tan variadas que implementar un blockchain para cada una de ellas sería algo inviable. Ethereum surge como un blockchain que entiende un lenguaje de programación de propósito general y sobre el que se pueden implementar todas las funcionalidades que se puedan codificar en este lenguaje.

Como dijo el creador de Ethereum Vitalik Buterin, "La mayoría de blockchains se asemejan a una calculadora, son muy buenos en una cosa pero no hacen nada más." [10]

El blockchain de Ethereum sin embargo no se asemeja a una calculadora. Es más parecido a un *smartphone*, donde se aporta la infraestructura necesaria para que la complejidad de desarrollar una aplicación resida en escribir el código de esta. Ethereum es un blockchain diseñado para abstraer y simplificar el desarrollo de aplicaciones blockchain.

2.1. Ethereum, lo más básico

Ethereum es un blockchain con bloques y transacciones como todos los demás al que le han añadido una serie de cosas:

1. Un lenguaje de programación integrado. Para implementar una aplicación lo que se debe hacer es escribir sus reglas en este lenguaje y subirlas al blockchain de Ethereum donde cualquiera podrá interactuar con ellas.
2. Dos tipos de cuentas. Cuentas de usuario que son controladas por un elemento externo al blockchain y smart contracts que son cuentas controladas por código.

Crear y utilizar una cuenta de usuario en Ethereum no es distinto de como se hace en el resto de blockchains.

Estado e historia:

En Bitcoin el estado se define de una manera simple: cuánto dinero tiene cada usuario en un momento dado.

En cambio, dentro de Ethereum se entiende como estado a la información actual para todas las cuentas que tiene el blockchain. El estado lo forman las variables que aparecen en la figura 15.

- *Balance*: cantidad de *ether* tiene una cuenta.
- *Nonce*: número de transacciones que ha emitido una cuenta.
- *Address*: dirección con la que se identifica una cuenta.
- *Code*: código que controla la cuenta, si se trata de una cuenta de usuario no contiene nada.
- *Storage*: región de memoria que el código puede utilizar para guardar variables no volátiles. Es "propiedad" del smart contract en el que está y ningún otro puede leer o escribir en esta región de memoria.

En Ethereum se denomina cambio de estado al proceso por el cual una cuenta altera alguno de estos valores.

La historia son el conjunto de sucesos que han hecho que se pase de un Estado a otro.

El siguiente código muestra una cuenta de Ethereum y sus atributos.

```
1    Tipo Cuenta
2          Balance;
3          Nonce;
4          Address;
5          Code;
6          Storage;
7    Ftipo
```

Figura 15: estructura de una cuenta en Ethereum.

2.2. Mensajes y transacciones

El término transacción se utiliza para referenciar al paquete que contiene un mensaje, el emisor, la firma del emisor, el nonce de este y el receptor. Toda transacción debe especificar quién es el receptor a no ser que se trate de una transacción que creará un smart contract, en cuyo caso el campo receptor se deja vacío.

Si el receptor de la transacción es una cuenta de usuario entonces el objetivo de dicha transacción es mover *ether*.

Por otro lado si el receptor de la transacción es un smart contract entonces su código se ejecutará. Durante su ejecución el código puede:

- enviar *ether* a otras cuentas
- leer y escribir en *storage*
- llamar a otros smart contracts, pudiendo provocar que su código se ejecute.

En Ethereum se entiende por mensaje al paso de información/*ether* de una cuenta a otra. Un mensaje en Ethereum puede haber sido creado por una entidad externa o por un contrato. Por otro lado los mensajes pueden contener datos y si el receptor del mensaje es un contrato este puede devolver una respuesta, abarcando así el concepto de llamada a una función.

Cuando se ejecuta una transacción se realizan los siguientes pasos:

1. Comprobar que la transacción está bien formada. La firma es válida y el nonce concuerda con el de la cuenta emisora. Si no, devuelve error.
2. Calcular la comisión del minero, *fee*, de la transacción y restarlo del balance de la cuenta. Incrementar el nonce. Si la cuenta no tiene suficiente balance para cubrir el fee, devuelve un error.
3. Transferir el *ether* mandado de una cuenta a otra, si la cuenta receptora no existe crearla. Además, si el receptor es un contrato se debe ejecutar el código pertinente hasta su finalización o hasta que la ejecución se quede sin gas.
4. Si la ejecución del código ha fallado o no se ha podido transferir el *ether* indicado todo vuelve al estado inicial menos la comisión, *fee,* pagado que se ingresa en la cuenta del minero.
5. Si la transacción se completó con éxito, enviar el fee al minero y devolver el gas restante al emisor.

En los siguientes apartados se verá con más detalles el concepto de gas.

2.3. Smart contracts

Para crear un smart contract dentro de Ethereum lo primero que se debe hacer es escribir las reglas que guiarán el comportamiento del smart contract. El código normalmente se escribe en un lenguaje de alto nivel como Solidity que luego es compilado al lenguaje que ejecuta Ethereum. Una vez compilado el código, este se incluye dentro de una transacción que se envía al blockchain y al ejecutarse creará un smart contract.

Tanto las cuentas de usuario como los smart contracts tienen la estructura presentada en el figura 15. *Address* representa “la dirección” en la que se encuentra un contrato. Para ejecutar una transacción sobre un smart contract esta debe ir destinada a su *address*. Una transacción dirigida a un smart contract debe contener el nombre de la función que se desea ejecutar y los datos que debe usar en su ejecución.

Como se ha expresado en el apartado anterior, el receptor de una transacción puede ser un smart contract. Cuando esto sucede la transacción tiene que indicar el nombre de la función que debe ejecutarse. Podría ser qué al procesar una transacción este nombre no coincidiera con ninguna de las funciones del smart contract.

Cuando esto ocurre el smart contract ejecutará una función llamada *función de fallback* [7]. Solo puede haber una función de este tipo por smart contract y no tienen nombre ni parámetros.

2.4.Ejecución del código

El código de los smart contracts de Ethereum está escrito en un lenguaje bytecode de bajo nivel basado en pila llamado EVM code. El código son una serie de bytes donde cada byte representa una operación. La ejecución es un bucle que de manera repetitiva ejecuta la operación indicada por el contador del programa e incrementa el contador. La ejecución termina cuando se produce o un error o se encuentra la instrucción STOP o RETURN [7].

Las instrucciones de este lenguaje son similares a las de otros sistemas basados en máquina virtual como el bytecode de Java.

El código puede utilizar tres tipos de entornos para almacenar información:

1. La pila, en ella se pueden apilar y desapilar valores de 32 bytes.
2. Memoria local, un array de bytes con direcciones de 256 bits.
3. *Storage*, en ella se guarda la información a largo plazo. La pila y la memoria se resetean cuando la ejecución de un mensaje termina; sin embargo la información en *storage* es persistente.

Es importante recalcar que todas las variables que un smart contract almacena en *storage* son visibles por cualquier usuario que tenga una copia del blockchain.

2.5. Gas

El lenguaje de los smart contracts es Turing completo. Esto alberga la ventaja de poder incluir una gran variedad de procedimientos dentro de un contrato. Sin embargo los lenguajes Turing completos traen consigo la indecibilidad del problema de parada que postula que, dado un programa escrito en un lenguaje Turing completo, es imposible determinar si este terminará su ejecución en algún momento.

Si la ejecución de un smart contract puede no terminar, y es imposible determinar si lo hará o no, entonces el blockchain de Ethereum corre el peligro de quedar bloqueado si tiene que ejecutar un smart contract que no termina.

Para evitar esto en Ethereum se utiliza la noción de *gas*, que es un modo de medir pasos computacionales. Cuando se envía una transacción se indica el *gas* disponible para realizar pasos computacionales durante su ejecución. Las instrucciones bytecode tienen un coste en gas

y la ejecución del código de un contrato producirá un error si se ha quedado sin gas. De esta manera se puede asegurar que independientemente del código que se ejecute en una transacción, en algún momento la ejecución terminará.

Uno podría preguntarse por qué Ethereum utiliza *gas* para medir pasos computacionales y no tiempo. Al fin y al cabo quien envía una transacción podría indicar en ella cuánto tiempo tiene para ejecutarse. El motivo por el que no se utiliza este mecanismo es porque la ejecución de una transacción dejaría de ser determinista ya que depende del computador en el que se ejecute la transacción. Al utilizar *gas* la ejecución de una transacción se mantiene determinista ya que es independiente del computador que la ejecute.

El *gas* en Ethereum puede definirse como un método determinista de medir recursos computacionales.

2.6.Reflexión

El cambio que plantea Ethereum recuerda en cierto modo al cambio que sufrieron los computadores en su momento.

Durante un tiempo los computadores se construían con un propósito fijo de tal manera que el software y el hardware se construían a la par para dicho propósito exclusivamente.

Fue más tarde cuando aparecieron los computadores programables capaces de realizar las tareas que se les indicase mediante el software que cualquiera podía desarrollar.

Del mismo modo la tecnología blockchain puede verse como el hardware que proporciona un entorno con unas características determinadas. Antes de Ethereum, para dar un uso distinto a la tecnología blockchain era necesario empezar casi de cero o utilizar un blockchain que no había sido construido para que se implementaran cosas sobre él. Ethereum, sin embargo, representa un blockchain de propósito general sobre el cual pueden implementarse nuevos protocolos que sirvan al propósito que el desarrollador quiera.

Conclusiones de la parte 1

En este punto del documento ya se han estudiado los distintos elementos involucrados en el desarrollo y funcionamiento de aplicaciones descentralizadas.

El estudio de la tecnología blockchain muestra que la interacción con una aplicación descentralizada es distinta a la interacción, por ejemplo, con una web alojada en un servidor. En el caso de las aplicaciones descentralizadas una interacción requiere la emisión de una transacción y su posterior inclusión dentro de un bloque minado. Como se vió en el capítulo 2, minar un bloque es muy costoso, lo que provoca que interactuar con una aplicación descentralizada en Ethereum tarde unos 15 segundos.

El tiempo que tarda un cambio producido en una aplicación descentralizada en ser válido limita los casos de uso de aplicaciones descentralizadas que tienen sentido. Aquellas aplicaciones que se benefician de la interacción en tiempo real con un servidor no tienen sentido dentro de un blockchain actualmente ya que perderían los beneficios de la interacción en tiempo real.

La noción de gas es determinante a la hora de conocer qué se puede implementar como aplicación descentralizada y qué no. A pesar de que el lenguaje de Ethereum sea Turing completo hay cómputos que no tiene sentido que una aplicación descentralizada realice. Cuando un cómputo necesita más gas del que la transacción le permite, este debe repartirse en varias transacciones. Como el tiempo que tarda una transacción en producir un resultado son 15 segundos, Ethereum es un medio demasiado ineficiente como para realizar grandes cómputos.

El coste en gas varía en función de la operación que se ejecute, cabe destacar el alto precio supone almacenar información permanente en Ethereum (200 gas por cada palabra de 256 bits leída y 20.000 por cada palabra de 256 bits almacenada). Esto provoca que los cómputos que realicen un gran número de operaciones sobre el almacenamiento permanente tengan grandes posibilidades de necesitar más de una transacción para completarse.

Teniendo en cuenta que el gas cuesta *ether*, crear una aplicación que realiza cómputos demasiado largos o almacena grandes cantidades de información en el blockchain provocará que su uso sea demasiado caro como para tener sentido.

Por otro lado, desde un punto de vista de escalabilidad, implementar una aplicación en Ethereum con cómputos que requerirán más gas cuantos más usuarios tenga sería implementar una aplicación abocada al fracaso, ya que su coste en gas aumentará hasta dejar de ser viable.

Como conclusión, Ethereum, aunque se trate de una máquina virtual que entiende un lenguaje Turing completo, tiene un límite muy marcado en el tipo de aplicaciones que puede almacenar. Esto se debe a que la tecnología blockchain actualmente es muy ineficiente. Las aplicaciones que actualmente tienen sentido en Ethereum son aquellas que realizan cómputos poco costosos y que almacenan poca información en el blockchain. Además su finalidad debe

aprovecharse del carácter descentralizado del blockchain ya que de lo contrario, al ser un entorno de ejecución tan ineficiente, no tiene sentido implementarlas en Ethereum.

Parte 2

Introducción a la parte 2

En la parte 1 se concluye que la utilidad de las aplicaciones descentralizadas es la de servir de intermediarias en procesos simples que no requieren mucha memoria ni cómputos costosos.

En este apartado se comprobará cómo afecta a una aplicación descentralizada no respetar los límites marcados en la parte 1. Se desarrollará una aplicación descentralizada que sobrepase estos límites.

Este apartado también se utilizará para mostrar los servicios que se utilizan en el diseño de aplicaciones descentralizadas. El lector conocerá lo que necesita utilizar en de caso de querer crear sus propias aplicaciones descentralizadas.

3.Infraestructura para el desarrollo de DApps en Ethereum

Cómo se desarrolla actualmente una DApp dentro de Ethereum

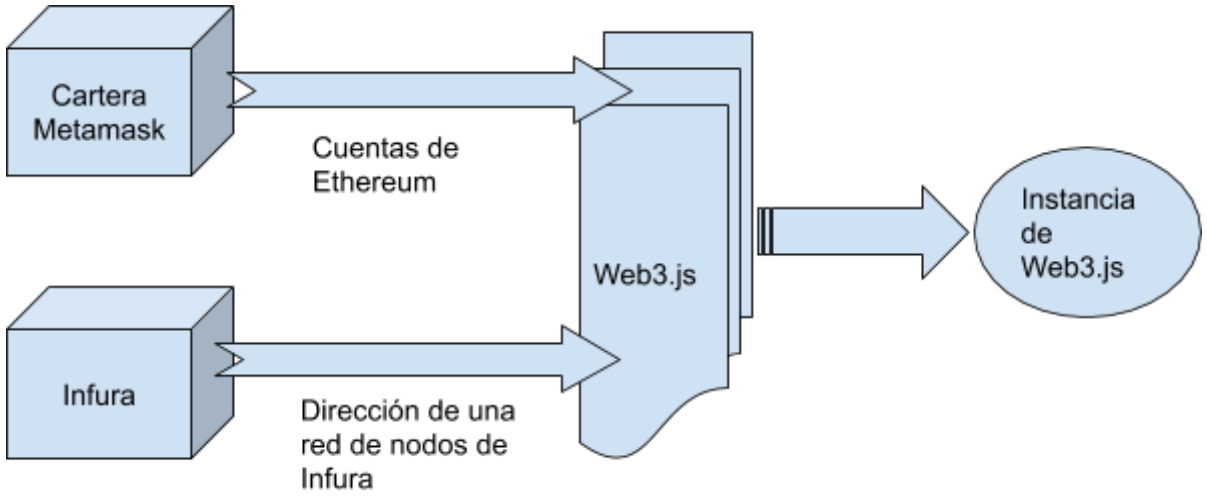


Figura 16: esquema de la creación de una instancia de Web3.js

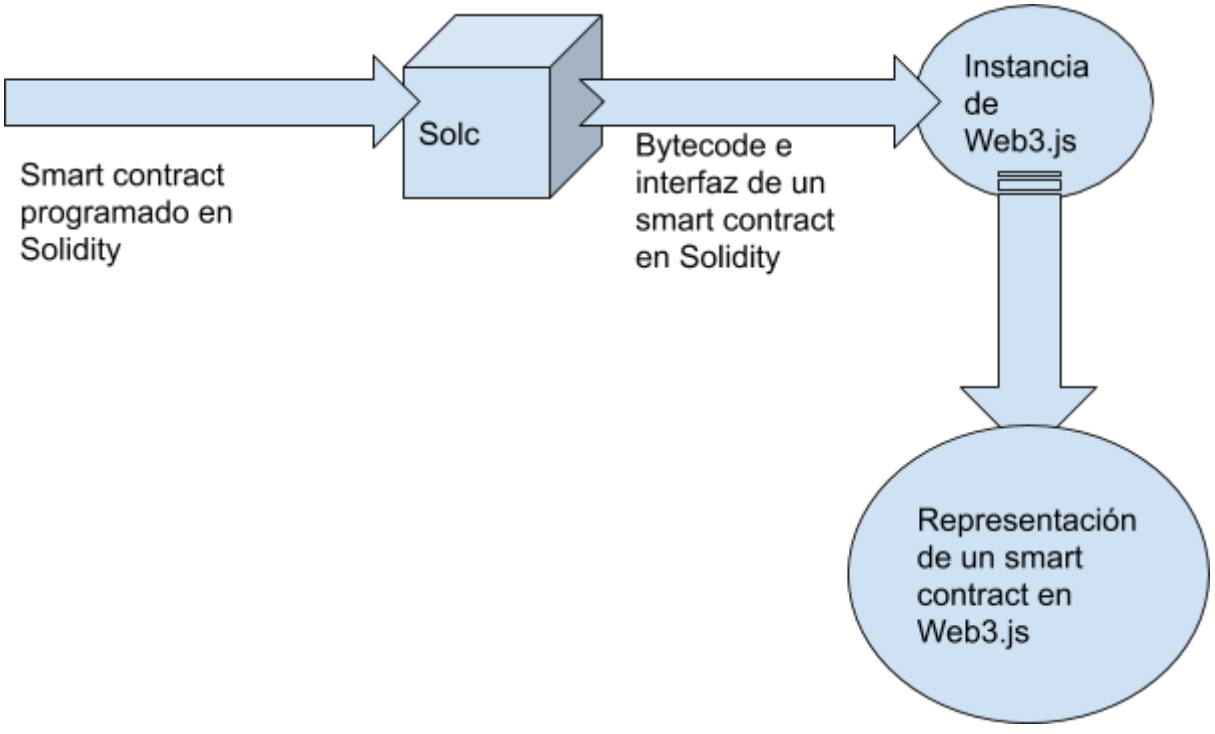


Figura 17: creación de un objeto que en Web3.js representa a un smart contract.

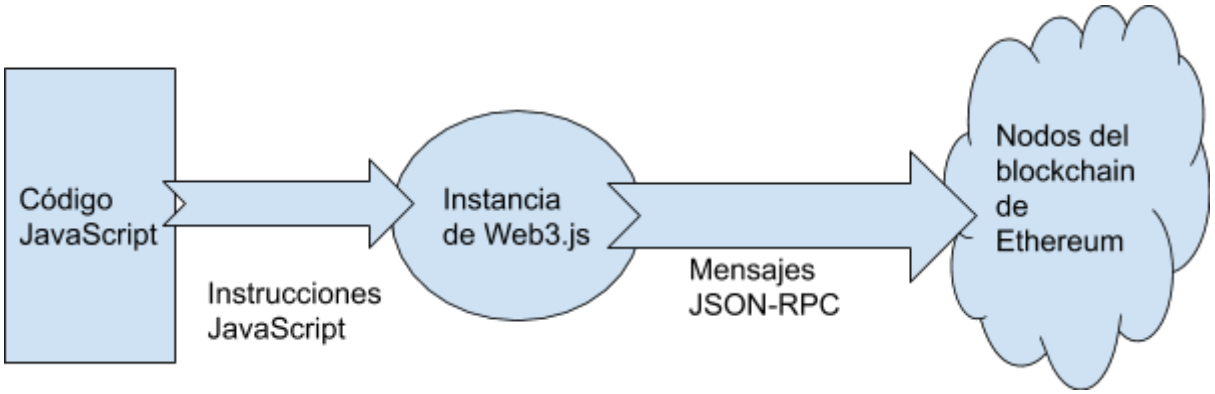


Figura 18: interacción del código Javascript con un nodo de Ethereum a través de Web3.js

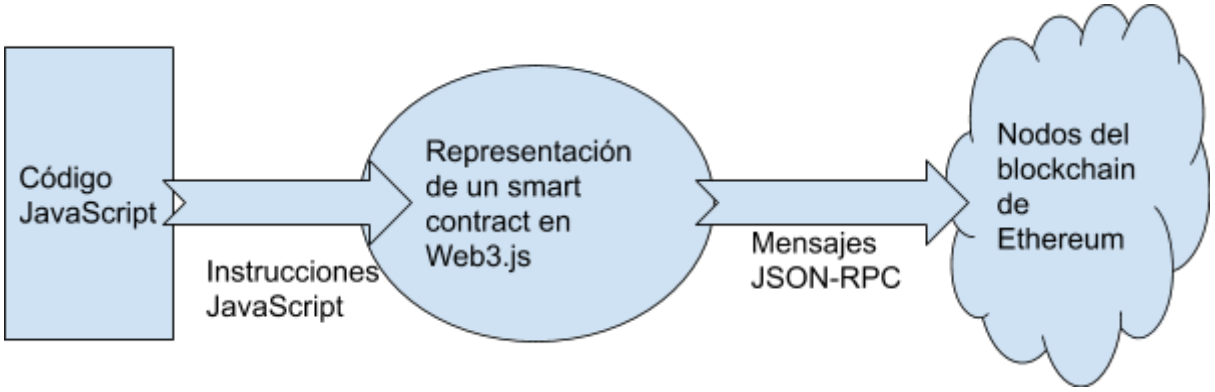


Figura 19: interacción con un smart contract en Ethereum a través de su representación en Web3.js

En este capítulo se expondrá la infraestructura utilizada para desarrollar la aplicación descentralizada Galactic Chain. En las figuras 16, 17, 18 y 19 aparecen los distintos servicios que forman esta infraestructura y cómo interactúan entre sí. Al final de este capítulo se espera que el lector comprenda, aunque de modo superficial, cuál es la finalidad de estos servicios.

3.1.¿Qué se necesita para desarrollar una aplicación descentralizada?

En primer lugar se necesita un lenguaje de alto nivel capaz de ser compilado a bytecode de la máquina virtual de Ethereum. Cuanto más robusto sea este lenguaje y más se abstraiga del código de bajo nivel de la máquina virtual de Ethereum, más sencillo será el desarrollo de smart contracts.

Por otro lado se necesita de algún mecanismo que permita probar los smart contracts diseñados. Subir un sistema al blockchain de Ethereum e interactuar con él cuesta *ether*, además del tiempo de espera necesario entre la emisión de una transacción y su ejecución. Es por esto por lo que se necesita de alguna herramienta que permita probar smart contracts antes de subirlos al blockchain de Ethereum.

Por último es necesario que en los lenguajes con comunidades más grandes, como C++ y Java, se desarrollen librerías que faciliten la interacción con los smart contracts de un blockchain. Esto permite que los smart contracts puedan ser utilizados por un público mayor.

3.2.Solidity, un lenguaje de alto nivel para la EVM

Existen distintos lenguajes de alto nivel capaces de compilar a código de bytecode de Ethereum, cada uno de ellos con cualidades que lo distinguen del resto. Los más famosos son:

1. Solidity: es el lenguaje de programación más conocido para Ethereum. Cuenta con un entorno de desarrollo en línea llamado Remix [22] desde el cual se pueden desarrollar, compilar y depurar smart contracts en distintas versiones del lenguaje.
2. Vyper: se trata de un lenguaje de programación orientado a contratos con un estilo semejante a Python.

A pesar de que Vyper sea un lenguaje que parece estar tomando popularidad, Solidity actualmente es el único lenguaje lo suficientemente robusto como para soportar el desarrollo de sistemas medianamente complejos.

Esta opinión se basa en el nivel en el que se encuentra la documentación de ambos lenguajes [21][23]. Por otro lado Solidity aporta la capacidad de utilizar la herramienta ya mencionada anteriormente, Remix, lo que facilita el desarrollo.

3.2.1.Solc, compilando código Solidity

Para crear un smart contract en Ethereum hay que enviar una transacción con el código de bytecode de dicho smart contract. Lo normal es desarrollar la lógica del smart contract en un lenguaje de alto nivel como Solidity y después utilizar un compilador que proporcione el código de bytecode que se puede incluir en una transacción (figura 17).

Para compilar los smart contracts desarrollados en Solidity se utiliza el compilador Solc. En la figura 20 se observa cómo se utiliza Solc desde Javascript para compilar el smart contract "Contract" con código "source". Tras compilar el código Solidity del smart contract, Solc, genera el código de bytecode del smart contract, siendo este el que debe incluirse en una transacción cuando se desee crear dicho smart contract en Ethereum.

```
1  const{interface, bytecode} = solc.compile(source,1).contracts[':Contract'];
```

Figura 20: instrucción Javascript capaz de compilar código Solidity a código de bytecode de Ethereum.

Por otro lado en la instrucción de la figura 20 Solc genera un elemento llamado *interface*. *Interface* es un fichero .json que representa el ABI (*application binary interface*) del programa. Ethereum no necesita este fichero para crear el smart contract, de hecho no se envía

en ningún momento. A partir del código de bytecode de un smart contract no se puede saber cuáles son sus funciones, qué parámetros reciben y qué visibilidad tienen. El fichero ABI contiene toda la información necesaria para interactuar con el smart contract.

3.3. Web3, una librería con funcionalidad para el ecosistema de Ethereum

El código de bytecode de un smart contract cumplirá con su propósito una vez se encuentre en el blockchain de Ethereum y para ello debe ser posible interactuar con dicho blockchain.

Web3 permite desarrollar clientes que interactúen con nodos del blockchain de Ethereum. A través de Web3 se puede, entre otras cosas, enviar *ether* de una cuenta de Ethereum a otra, crear smart contracts e interactuar con smart contracts ya existentes en el blockchain.

Un nodo en Ethereum solo entiende mensajes en un lenguaje llamado JSON-RPC [28], que no es demasiado legible. Un usuario puede escribir mensajes en un formato más amigable (como JavaScript) y enviarlos a un nodo a través de Web3, que se encargará de traducirlo a JSON-RPC. La figura 18 refleja este proceso cuando se interactúa con un blockchain, pero no con un smart contract. En la figura 19 sin embargo la interacción sí es con un smart contract del blockchain.

Actualmente la librería Web3 tiene una versión estable para JavaScript (Web3.js), Python (Web3.py) y Java (Web3J). Debido a la gran comunidad que hay formada alrededor de JavaScript, la versión más avanzada de Web3 es Web3.js.

En la figura 21 se encuentra el código necesario para compilar un smart contract y almacenarlo en Ethereum a través de JavaScript.

```
1  const {interface, bytecode} = solc.compile(source, 1).contracts[':Contract'];
2     contractResult = await new web3.eth.Contract( JSON.parse(interface) );
3  contractResult.deploy( { data: '0x' + bytecode } ).send( { from: accounts[0]} );
```

Figura 21: creación de un smart contract en el blockchain de Ethereum a través de Web3.

Al terminar su ejecución, el código de la figura 21 habrá creado un smart contract en el blockchain de Ethereum. Tras obtener el código de bytecode y el ABI en la línea 1, la línea 2 le pasa a Web3.js dicho ABI, de modo que Web3.js es capaz de crear un objeto JavaScript que representa dicho smart contract. Esto puede verse en la figura 17.

Una vez obtenida la representación del smart contract en JavaScript la línea 3 envía el código de bytecode de este a un nodo del blockchain de Ethereum que se encargará de incluirlo en el blockchain.

3.4.MetaMask e Infura

Web3.js permite la comunicación con los nodos del blockchain de Ethereum a través de JavaScript, sin necesidad de escribir mensajes JSON-RPC.

Sin embargo, para que el mensaje que se envía a través de Web3.js tenga algún efecto sobre el blockchain de Ethereum se debe tener un nodo que lo reciba. Para crear una instancia de Web3.js que proporcione acceso al blockchain de Ethereum se necesita un nodo al que enviar los mensajes JSON-RPC, como muestra la figura 18.

Infura proporciona acceso a un conjunto de nodos de Ethereum. De este modo los usuarios del blockchain de Ethereum no necesitan implementar un nodo, sólo deben conocer la dirección de los nodos de Infura con los que conectarse.

Los nodos conocen el estado global del blockchain y envían transacciones a los mineros, sin embargo un nodo no puede crear transacciones, solo las recibe y las emite.

El software encargado de crear dichas transacciones se denomina cartera (*wallet*). Este software posee las claves públicas y privadas que deben usarse para firmar las transacciones que modifican el estado del blockchain.

En este caso, como puede verse en la figura 16, se ha utilizado la cartera de Metamask debido a que es una cartera que se instala como extensión de Chrome y Firefox. Esto permite a los usuarios interactuar con los smart contracts diseñados a través de páginas web sin la necesidad de que estas almacenen sus claves públicas y privadas.

En la figura 22 se puede observar cómo se crea una instancia de web3 totalmente funcional. Metamask cuando detecta que el navegador esta en una web que utiliza Web3.js inyecta en la web un código semejante al de la figura 22, de este modo el usuario puede interactuar con la web libremente.

```
1  //El primer parámetro de denomina semilla y sirve como login a la //cartera.
2  //El segundo parámetro es la dirección de los nodos de Infura a los //que se
3  enviarán mensajes JSON_RPC
4
5  const provider = new HDWalletProvider(
6         'arctic economy…..',
7        'https://rinkeby.infura.io/…..'
8  );
9  const web3 = new Web3(provider);
```

Figura 22 : creación de una instancia de Web3.js que utiliza una cartera de Metamask y la red de nodos de Ethereum de Infura.

3.5.Rinkeby, un Blockchain para probar smart contracts

Subir un smart contract al blockchain de Ethereum, así como interactuar con él requiere poseer una cuenta con *ether* suficiente para realizar estas transacciones. Actualmente el *ether* tiene un valor real en bolsa [38] lo que hace que probar un smart contract directamente en el blockchain de Ethereum sea excesivamente costoso.

Además, el tiempo que se tarda en ejecutar una transacción en el blockchain de Ethereum es demasiado grande y por lo tanto probar smart contracts complejos sería un proceso muy largo.

Otra característica a tener en cuenta es la imposibilidad de actualizar un smart contract. Cada vez que se quiere actualizar un smart contract habría que eliminarlo del blockchain de Ethereum y subir la nueva versión lo que costaría demasiado tiempo y *ether*. A esto se le suma el *ether* que costaría pasar la información contenida en el antiguo smart contract a la nueva versión.

Existe un tipo de blockchains de prueba que simulan al de Ethereum y cuya única finalidad es proporcionar un entorno de pruebas más eficiente. En los blockchains de prueba el protocolo no asegura que el blockchain esté totalmente a prueba de ser alterado lo cual hace que utilizar la palabra blockchain para referirse a ellos sea incorrecto. Aun así, desde el punto de vista del desarrollador que quiere probar sus smart contracts, suponen un entorno bastante realista en el que poder realizar pruebas sin ningún riesgo.

Rinkeby es un blockchain de pruebas que simula el blockchain de Ethereum. Sus tokens se llaman *ether* también, lo cual no quiere decir que tener 1 *ether* en Rinkeby equivalga a tener dinero real, de hecho 1 *ether* de Rinkeby no tiene ningún valor ya que lo que da valor al *ether* es el blockchain de Ethereum. El *ether* de Rinkeby no puede convertirse en *ether* de Ethereum ni viceversa. Infura tiene nodos que se conectan al blockchain de Rinkeby y las cuentas de Metamask tienen representación en Rinkeby, por lo que se amolda perfectamente al resto de servicios aquí presentados.

4.Galactic Chain: gestión y supervisión de rescates interplanetarios gracias a Ethereum

Un caso de uso que excede los límites actuales de las aplicaciones descentralizadas.

El caso de estudio que se plantea en esta capítulo no tiene como objetivo ser un problema real cuya solución tendrá una repercusión directa en el mundo. El problema a plantear esta diseñado para que su solución exceda algunos de los límites encontrados en la primera parte de este TFG. Exponiendo así los problemas de escalabilidad que esto implica.

4.1.Contexto

Este capítulo cuenta un problema ocurrido en un futuro en el que la especie humana es una especie interplanetaria. El planeta Tierra se volvió inhabitable y, como la tecnología lo permitía, la especie humana decidió habitar otros planetas del sistema solar.

La especie está repartida en el espacio entre los planetas que se observan en la siguiente imagen.

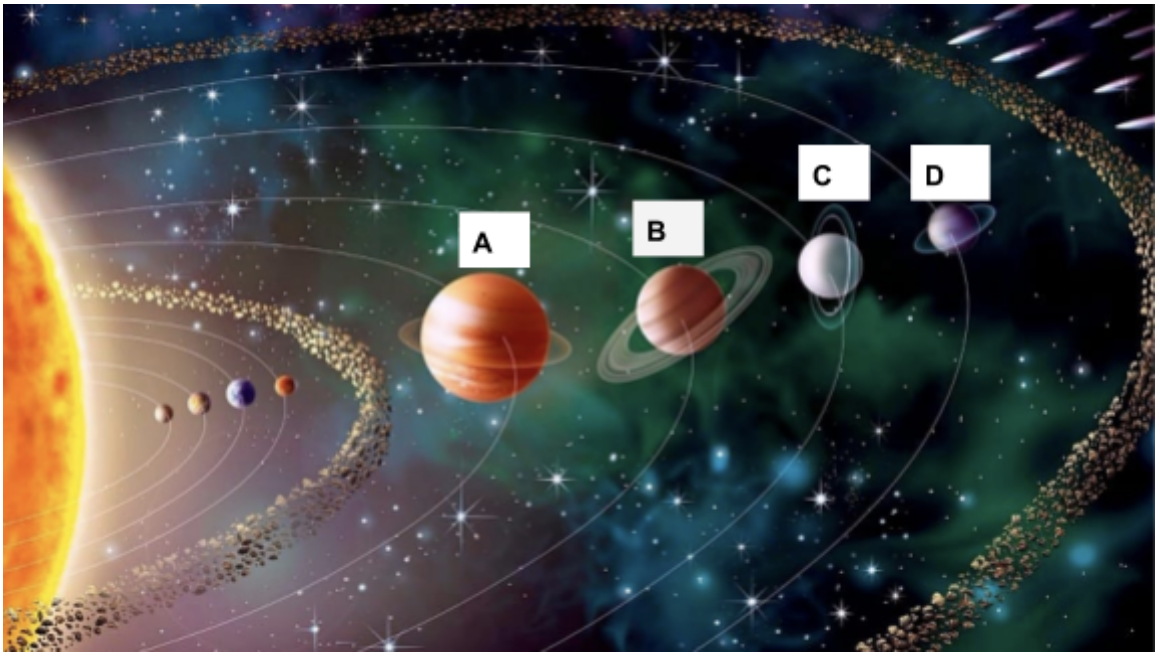


Figura 22: sistema solar

La especie lleva el tiempo necesario esparcida entre los planetas dados como para que se hayan creado organismos encargados de mediar en los problemas interplanetarios. Estos organismos están formados por individuos de cada uno de los planetas.

4.2.Problema

Los viajes interplanetarios entre los planetas planteados es algo común. A pesar de que estos viajes son algo fundamental, al tribunal de justicia interplanetario le están planteando grandes quebraderos de cabeza.

Actualmente el rescate de las naves que quedan a la deriva durante un viaje interplanetario no está regulado. Los planetas interesados en que la nave sea recuperada y los equipos de rescate suelen tener que llegar a acuerdos poco fiables y regulados.

El tribunal de justicia interplanetario cada vez tiene más casos en los cuales el rescate ha terminado en disputa entre las dos partes. Los dos casos que están dando problemas al tribunal de justicia son los siguientes:

1. Supongamos dos planetas de los presentados anteriormente, *A* y *B*. *A* ha perdido una de sus naves y ha prometido pagar una gran cantidad de dinero a quien la rescate. Un equipo de rescate enviado por *B* se las ha ingeniado para rescatar la nave y la ha llevado al planeta *A*. El problema ha surgido cuando *A* ha decido no pagar al equipo de rescate de *B* obligando a éste a llevar el caso al tribunal de justicia interplanetaria.
2. Los mismos planetas del primer caso han tenido ahora otro problema, esta vez ha sido el equipo de rescate enviado por *B* quien, al ver que lo que contenía la nave era muy valioso, la ha saqueado. Una vez saqueada, ha llevado la nave al planeta *A* donde ha cobrado el dinero del rescate. Cuando en *A* se han percatado del saqueo han decidido llevar el caso al tribunal de justicia interplanetaria.

El tribunal de justicia interplanetaria, cada vez que se le presenta uno de estos casos, tiene que llevar a cabo una investigación que implica a dos planetas distintos y por lo tanto consume demasiados recursos.

Al tribunal le gustaría poder contar con un sistema informático que registrara todo el proceso que siguió el rescate, desde los precios acordados hasta las condiciones en las que se encontraba la nave perdida. En general, este sistema informático serviría de una vez por todas para regular estos rescates.

Una opción que se ha planteado es la de repartir servidores que den soporte al sistema entre todos planetas. Al tribunal no le convence esta opción ya que da demasiado control a los planetas sobre el sistema y, teniendo en cuenta que las disputas entre planetas son algo muy común, sería un sistema poco fiable.

Klever es un miembro del tribunal que ha llegado a la conclusión de que, debido a las tensiones entre planetas, la especie humana solo se fiará del sistema si es descentralizado. Muchos de los equipos de rescate y de los dueños de naves tienen conocimientos de informática avanzada por lo que el sistema debe estar a prueba de manipulación. Esta conclusión le hace recordar que el ser humano unos cuantos años antes de que la especie se volviese interplanetaria

inventó una tecnología que permitía mantener información de manera descentralizada sin arriesgar que esta pueda ser manipulada, la tecnología *blockchain*.

Klever procede a investigar acerca de cómo ha evolucionado esta tecnología desde entonces. Resulta que existe un blockchain llamado Ethereum, que lleva muchos años funcionando, diseñado para que sus usuarios pudiesen implementar aplicaciones descentralizadas en él. Parece que Klever ha encontrado una tecnología en la que implementar el sistema que el tribunal necesita.

4.3.Terminología

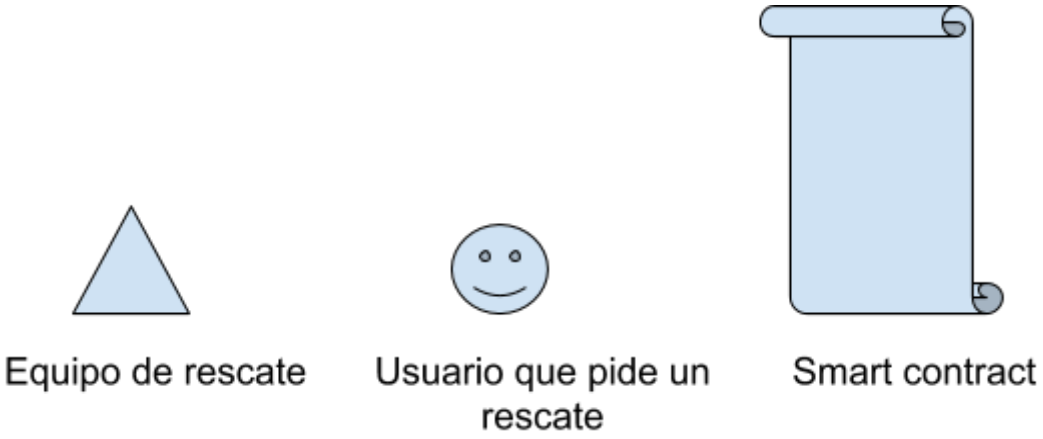


Figura 23: símbolos utilizados en los diagramas

Definiciones:

- Equipo de rescate: conjunto de personas dedicadas a encontrar y llevar con sus dueños las naves desaparecidas.
- Nave desaparecida: aquella que ha quedado a la deriva durante un viaje interplanetario. Puede contener objetos en su interior que también se quieren recuperar.
- Rescate: búsqueda y posterior entrega de una nave desaparecida a su dueño.
- Recompensa: cantidad de tokens que el equipo de rescate recibirá si entrega una nave desaparecida con los objetos que el dueño indicó que se encontraban en ella.
- Alerta: mensaje que indica la existencia de una nave desaparecida e informa de la recompensa que se ofrece por su rescate.
- Firma de una nave: secuencia de caracteres utilizada para que un equipo pruebe que ha encontrado una nave. Cada vez que una nave desaparece su dueño puede activar remotamente un sistema que genera una secuencia de caracteres nunca antes utilizada y la muestra en la pantalla de control de la nave desaparecida. Debido a que la secuencia se renueva cada vez que se pierde la nave, un equipo de rescate solo puede conocerla si ha encontrado la nave.

- Apuntarse a un rescate: pagar el precio de indicar que se va a intentar rescatar la nave desaparecida de un rescate determinado.
- Aceptar una alerta: ser el primero en indicar que se va a intentar rescatar la nave de una alerta. La alerta se convierte en un rescate y el precio es el mismo que apuntarse a un rescate.
- Firmar un rescate: aportar la firma de la nave desaparecida en dicho rescate.

4.4.El sistema informático

El tribunal ha reunido a un grupo de ingenieros informáticos para diseñar e implementar este sistema en *Ethereum*. La idea que mueve este sistema es la de que los acuerdos sobre rescates de naves se diseñen y ejecuten en la máquina virtual de *Ethereum* que no puede ser manipulada.

El equipo de informáticos ha realizado un estudio para saber qué es lo que más le preocupa más a cada una de las partes del rescate. Los resultados están en la siguiente imagen:

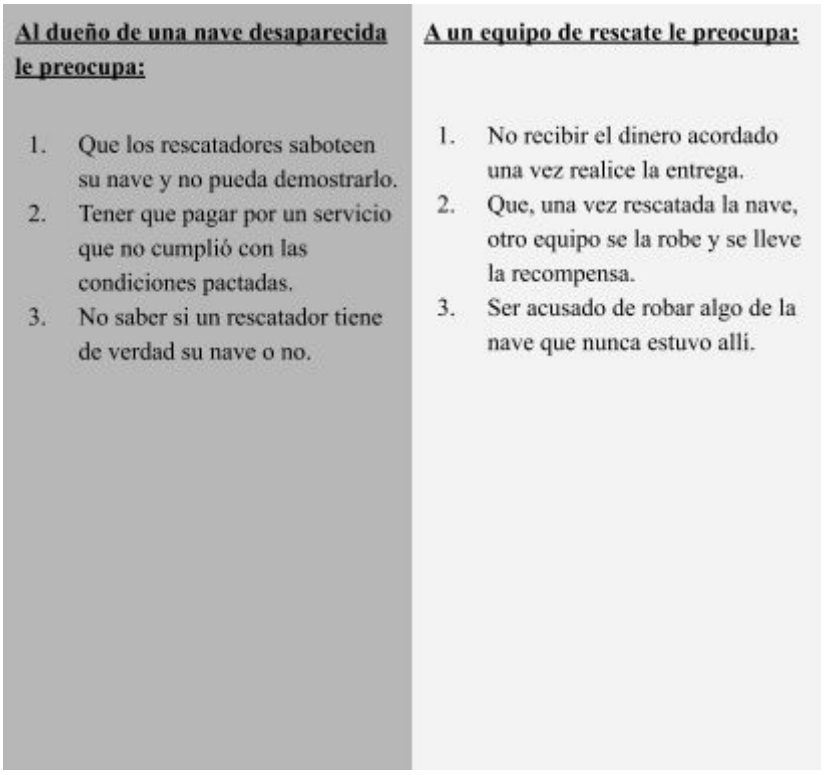


figura 24: preocupaciones de los dueños de naves desaparecidas y de los equipos de rescate

El sistema debe conseguir que las preocupaciones de la figura 3 dejen de ser un problema durante el rescate de naves desaparecidas.

4.4.1. Interacción de los usuarios con el sistema y visión general

En la siguiente tabla se presentan las distintas maneras en las que un usuario puede interactuar con el sistema.

Función	Quién la llama	Descripción
crear_alerta(D, S, R) -D: descripción de una nave desaparecida. -S: firma de la nave desaparecida. -R: recompensa por el rescate.	El dueño de una nave desaparecida.	El sistema registra que ha desaparecido una nave con la descripción D, la firma S y la recompensa R. El sistema registra esta nave dándole un identificador a la alerta
eliminar_alerta(Id) -Id: identificador de una alerta.	El dueño de la nave desaparecida de la alerta Id.	Si quien llama a la función es el creador de Id entonces la alerta se elimina y la recompensa se envía de vuelta al creador de la alerta.
Aceptar_alerta(Id) -Id: identificador de una alerta.	Un equipo de rescate.	Indica que va a intentar rescatar la nave desaparecida de la alerta Id. Como prueba de que efectivamente intentará rescatar la nave llamar a esta función cuesta una cantidad fija que se incluirá dentro de la recompensa y que obtendrá junto con ella quien encuentre la nave. En este punto la alerta pasa a llamarse rescate

Participar_Rescate(Id) -Id: identificador de un rescate.	Un equipo de rescate.	En este caso Id representa un rescate, es decir, ya hay un equipo de rescate intentando recuperar la nave desaparecida en Id. El equipo de rescate que llama a esta función indica que va a competir por la recompensa de Id con los que ya están intentando recuperar la nave desaparecida. La llamada a esta función cuesta una cantidad fija que se incluye en la recompensa del rescate y que obtendrá junto con ella quien encuentre la nave.
firmar_rescate(Id, S) -Id: identificador de un rescate. -S: supuesta firma de la nave -desaparecida en Id.	Un equipo de rescate que participa en el rescate Id, que aún no se ha firmado.	Un equipo de rescate que participa en Id indica que ha encontrado la nave. Para probar esto aporta la firma de la nave desaparecida, S. Si la firma es correcta el rescate se ha firmado y el equipo que ha llamado a la función queda registrado como ganador de la recompensa. Si no es correcta la llamada no tiene efecto.
recompensar_rescate(Id) -Id: identificador de un rescate firmado.	El dueño de la nave desaparecida en Id.	Si el rescate Id ya ha sido firmado entonces se le envía la recompensa al equipo de rescate registrado como ganador. A continuación el rescate se elimina después

		del sistema. Si Id aún no ha sido firmado, la llamada no tiene efecto.
incluir_objeto_desaparecido(Id,D) -Id: identificador de un rescate -D: identificador del objeto desaparecido.	El dueño de la nave desaparecida en Id	Si Id aún no se ha firmado entonces se registra que en la nave desaparecida de Id está el ítem con descripción D.

En la figura 28 se muestran las funciones separadas en dos grupos, las que utilizarán los dueños de naves desaparecidas y las que utilizarán los equipos de rescate.

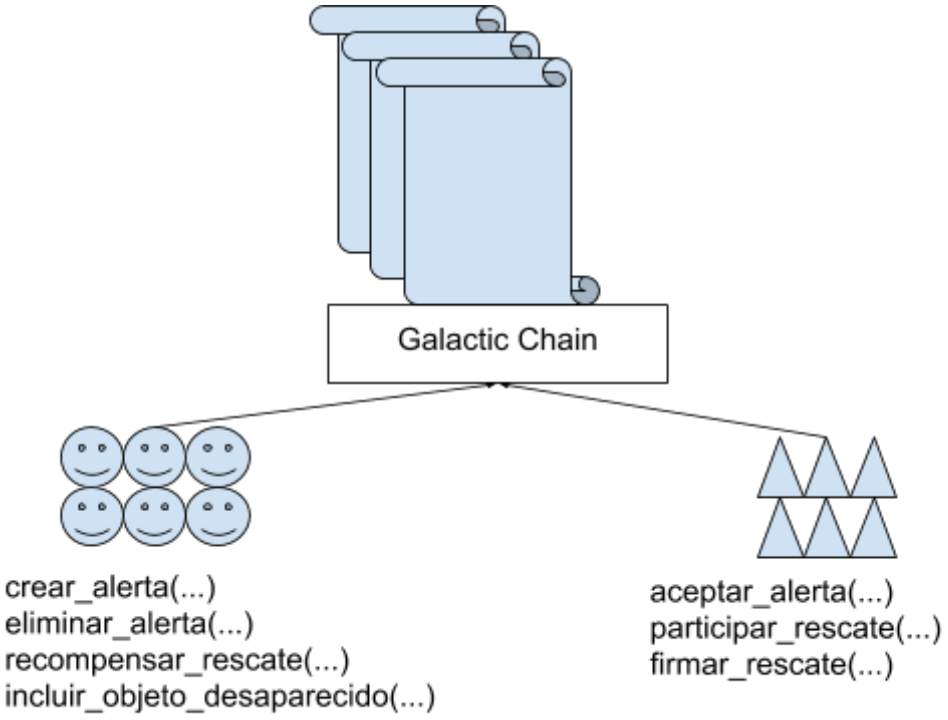


Figura 25: esquema general

4.4.2.Estados por los que pasa una nave desaparecida y sus transiciones

En la figura 26 se muestra la transición, de una nave desaparecida, de que no conste su transición en Galactic Chain (estado 0) a que una alerta de Galactic Chain informe de su

desaparición (estado 1). Esta transición la provoca el dueño de la nave desaparecida al crear una alerta.

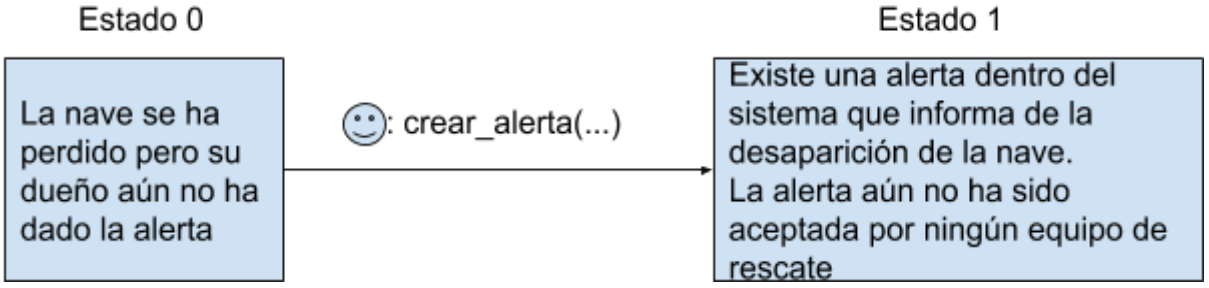


Figura 26: transición del estado 0 al estado 1

La figura 27 muestra la transición de que la desaparición de una nave sea una alerta (Estado 1) a que sea un rescate que no se ha firmado (estado 2). Esta transición la provoca un equipo de rescate al ser el primero que indica sus intenciones de encontrar la nave pagando el coste de aceptar una alerta. Mientras un rescate esté en el estado 2 el dueño podrá indicar qué objetos, de los que hay dentro de dicha nave, quiere recuperar. Por otro lado los equipos de rescate pueden competir por la recompensa del rescate mientras este esté en el estado 2, solo deben apuntarse al rescate, pagando el coste de realizar dicha acción.

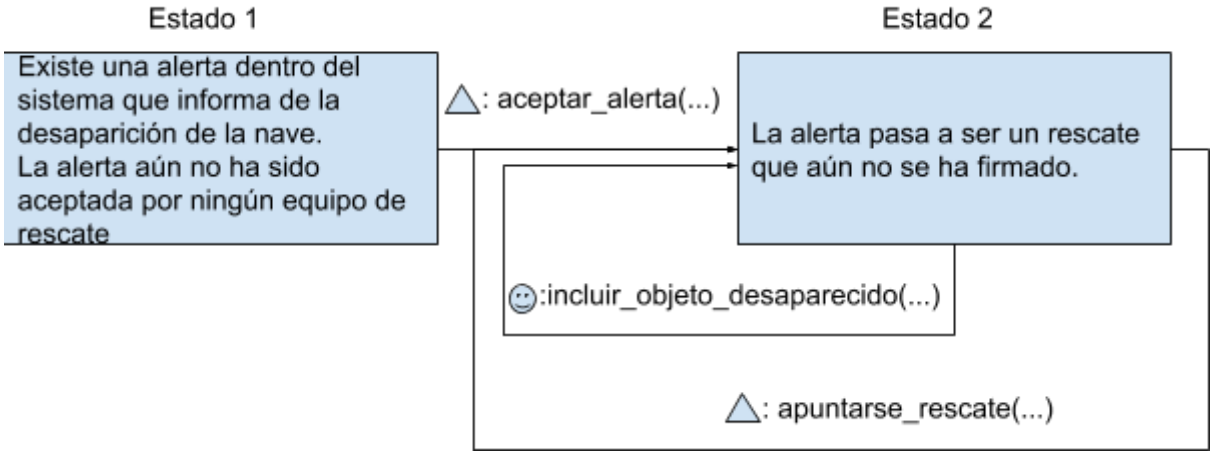


Figura 27: transición del estado 1 al estado 2

En la figura 28 un rescate pasa de no estar firmado (estado 2), a estar firmado (estado 3). En el estado 3 de un rescate el dueño de la nave ya no puede informar de que hay objetos en la nave que quiere recuperar. Del mismo modo, los equipos de rescate, que no están apuntados a este rescate, ya no podrán apuntarse. Para realizar esta transición un equipo de rescate, que estuviese apuntado al rescate en cuestión, debe firmar el rescate. La nave solo pasará del estado 2 al estado 3 si la firma proporcionada coincide con la que debería tener la nave.

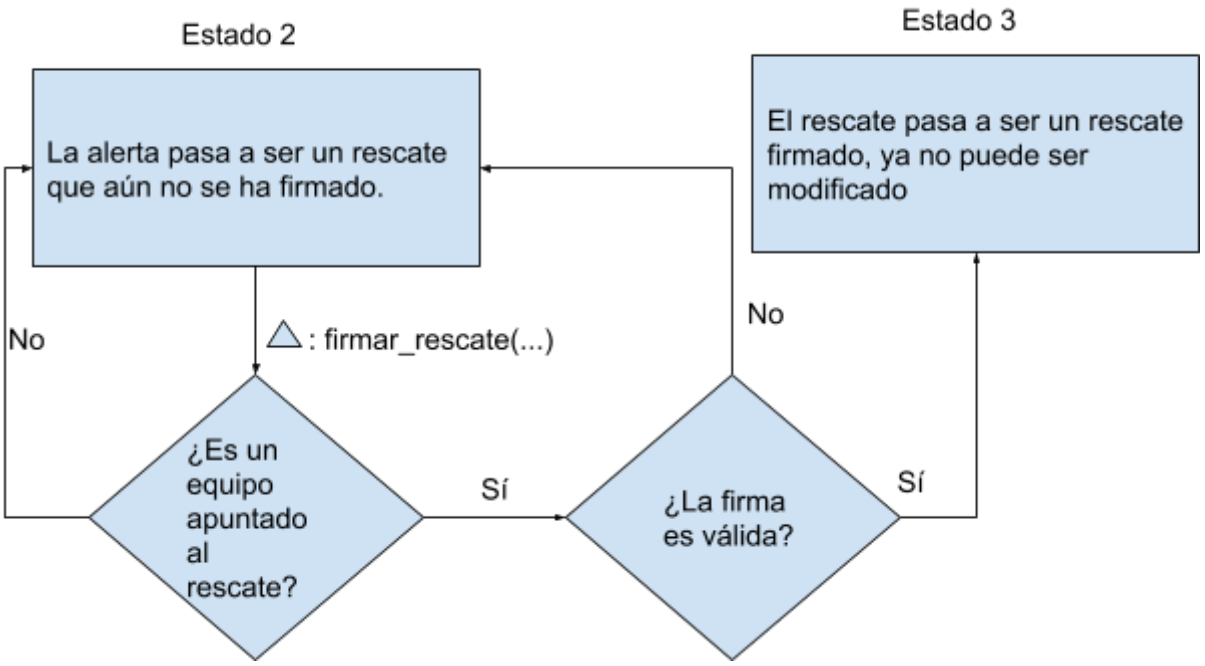


Figura 28: transición del estado 2 al estado 3

En la figura 29 la nave que ha desaparecido en un rescate que ha sido firmado (estado 3) es entregada al dueño. El dueño, al ver que todo esta en orden, envía a Galactic Chain la orden de recompensar el rescate, momento en el que el equipo de rescate recibe los tokens y el rescate es eliminado del sistema.

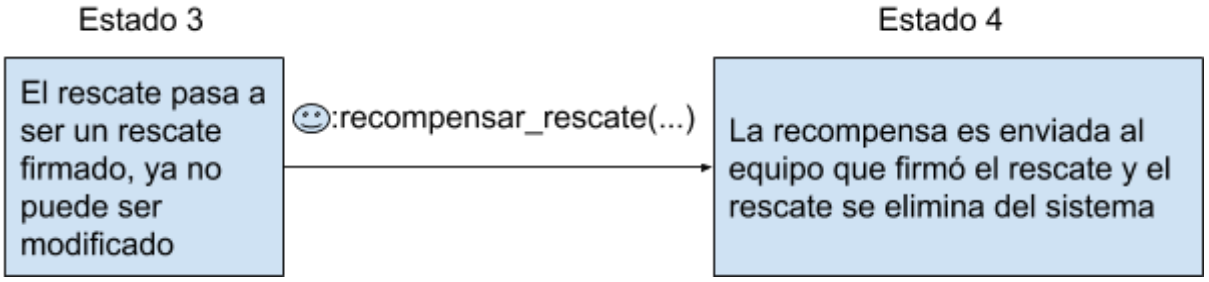


Figura 29: transición del estado 3 al estado 4

4.5.Cómo se resuelven las preocupaciones de los usuarios de este sistema

En este apartado se comprobará si la interacción con el sistema presentada en el apartado 4.4.1 elimina las preocupaciones de los usuarios presentadas en la figura 24.

Preocupaciones de quien pide un rescate:

1. *Que los rescatadores saboteen su nave y no pueda demostrarlo*: en la figura 27 se observa que el estado 2 es el único que permite incluir ítems de la nave

desaparecida. Esto significa que un equipo de rescate, antes de firmar el rescate, conoce todos los ítems que reclama el dueño de la nave y por lo tanto si alguno de estos ítems falta debe encargarse de justificarlo.

2. *Tener que pagar por un servicio que no cumplió con las condiciones pactadas*: como indica la figura 29, para recompensar un rescate el dueño de la nave es quien debe interactuar con el sistema. Dado que el equipo de rescate es el encargado de recuperar la nave con todos los ítems, debe justificar la ausencia de los que no estén si quiere obtener su recompensa.
3. *No saber si un equipo de rescate tiene de verdad su nave o no*: En la figura 28 se observa que, si el rescate ha pasado del estado 2 al ,3 es porque dicho rescatador ha firmado el rescate. Para realizar esta acción hay que tener la firma de la nave desaparecida, por tanto el dueño de la nave puede estar seguro de que el equipo de rescate tiene la nave.

Preocupaciones de los rescatadores:

1. *No recibir el dinero acordado una vez realice la entrega*: en cuanto se pide un rescate el dinero se ha enviado al sistema. La creación de una alerta implica el envío del dinero que se ofrece por el rescate.
2. *Que, una vez rescatada la nave, otro equipo se la robe y se lleve la recompensa*: firmar un rescate, no solo sirve para dar garantía de que la nave fue encontrada. También sirve para que el sistema registre al rescatador como tal y por lo tanto el único capaz de recibir la recompensa del rescate. El único que recibirá la recompensa cuando el rescate pase del estado 3 al 4 (figura 29) es el mismo equipo de rescate que hizo que el rescate pasase del estado 2 al 3 (figura 28).
3. *Ser acusado de robar algo de la nave que nunca estuvo allí*: del mismo modo por el que el dueño de la nave desaparecida puede demostrar que la nave ha sido saboteada, un rescatador puede demostrar que todo lo que se indicó que estaba en la nave antes de firmar el rescate sigue estando allí. La información del sistema es inmutable.

4.6.Implementación del prototipo en solidity.

El prototipo se ha implementado separando el trabajo en cuatro smart contracts, de este modo actualizar el sistema es una tarea más sencilla que puede realizarse de forma fraccionada. Los cuatro smart contracts que forman Galactic Chain son:

- Database: se encarga de registrar los *address* que pueden interactuar con el contrato.
- ContractList: implementa una lista doblemente enlazada de smart contracts. Permite eliminar y borrar de la lista sin consumir mucho gas.
- AlertControl: crea smart contracts que representan alertas y utiliza una instancia de ContractList para almacenarlos.

- Server: es el smart contract con el que se comunican los usuarios. Se encarga de identificar a los usuarios utilizando Database y de crear y destruir alertas utilizando ContractList. Además gestiona los rescates, ya que los almacena en *structs* de forma permanente.

En los siguientes apartados se detallan las estructuras de datos y las funciones de los contratos anteriores. En el repositorio de github [55] se puede encontrar el código completo de este prototipo.

4.6.1.Database

Debido a que se trata de un prototipo situado en Ethereum, cualquiera que tenga la dirección de alguno de los contratos puede intentar ejecutar alguna de sus funciones. Para evitar que un usuario ajeno al prototipo interactúe con él se introduce el smart contract Database, que debe identificar a todos los address que intentan interactuar con el prototipo.

El siguiente código muestra las variables que utiliza este smart contract. El diccionario *addressType* relaciona un address con el tipo de actor dentro de Galactic Chain (1 si es un equipo de rescate y 0 en otro caso). El diccionario *dataStore* almacena el nombre de dicho actor en hexadecimal. Por último, el diccionario *clean* indica si un address pertenece al prototipo o no.

Además de los diccionarios, Database cuenta con un booleano identificado como *deprecated* que indica si la la instancia esta desactualizada o no y con el *address* owner que es el de la cuenta que puede ejecutar las operaciones que modifican el estado del smart contract (añadir un usuario, eliminar la instancia Database y marcar la instancia como deprecated).

```
1  mapping(address => uint)  public addressType;
2  mapping(address=>bytes32) public dataStore;
3  mapping(address => bool)  public clean;
4  address owner;
5  bool deprecated;
```

Las siguientes cabeceras de función corresponden a las funciones de Database. En *deprecate, addData* y *kill* puede observarse el modificador *OnlyOwner*, esto significa que antes de que se ejecute la acción se realizará la comprobación *msg.sender == owner* y en caso de devolver *false* la transacción fallará. Las primeras 3 funciones son los *getters* de sus respectivos diccionarios. *Kill* se utiliza para eliminar el contrato y *deprecate* sirve para indicar que el contrato ya no tiene utilidad aunque no haya sido eliminado aún.

```
1  function isClean(address add) view  public returns (bool)
2  function getData(address add) view  public returns (bytes32)
3  function getType(address add)view  public returns(uint)
4  function deprecate() payable OnlyOwner public
5  function addData(uint Type,address id, bytes32 data) payable OnlyOwner public
6  function declareOwnerShip(address _owner) payable public
7  function kill() payable OnlyOwner public
```

4.6.2.ContractList

Este smart contract representa una lista doblemente enlazada de *address*, en él se utilizan los *address* como un puntero.

En el siguiente código se muestran las variables de este smart contract. Como toda lista enlazada, guarda un inicio, *ini*, y un address fuera de la lista al que permite determinar el último elemento, *end*, en este caso *end* es la dirección 0x00. El diccionario *isInTheStack* se utiliza para comparar si un address está en la lista o no. Por otro lado los diccionarios *nodeTonext* y *nodeToprev* son el address siguiente y anterior de uno dado. Por último la variable *Owner* sirve para identificar al dueño dentro de Ethereum.

```
1  address private ini;
2  address private end;
3  address private owner;
4  mapping(address => bool) private isInTheStack;
5  mapping(address => address) private nodeTonext;
6  mapping(address => address)  private nodeToprev;
```

Las cabeceras de las funciones de ContractList pueden verse en el siguiente fragmento de código. Son semejantes a las funciones de cualquier TAD que implemente una lista. La única que requiere más explicación es *declareOwnership*, esta función es llamada por el actual *owner* de la lista y pasa por parámetro el address del que será su nuevo *owner*.

```
1  function push(address n) payable public ;
2  function erase(address n) payable public ;
3  function top() view public returns(address);
4  function isEnd(address n) view public returns(bool) ;
5  function hasKey(address n) view public returns(bool) ;
6  function getNext(address n) view public returns(address);
7  function getPrev(address n) view public returns(address);
8  function declareOwnership(address _owner) OnlyOwner payable public
```

4.6.3. AlertControl

Este smart contract se encarga de almacenar en un ContractList las alertas del prototipo. Contiene como atributos la lista de alertas *list*, un diccionario de string a booleano que permite saber si una alerta existe ya o no, *bytes32_to_bool*, el número de alertas que ha registrado, *length,* y el address de su dueño del smart contract, *owner*. En el siguiente código se muestra la declaración de estos atributos.

```
1  address private owner;
2  address private list;
3  uint256 private length;
4  mapping (bytes32 => bool) bytes32_to_bool;
```

Las funciones de este smart contract pueden verse en el siguiente código. *NewAlert* se utiliza para crear una nueva alerta a partir de la descripción de esta. Por otra parte *verifyAlert* y *getValue* reciben el address de un smart contract que representa una alerta, en el primer caso la verifica, es decir, la borra de la lista y en el segundo devuelve la información de la alerta. *getPendingList* se utiliza para obtener un conjunto lo más grande posible de alertas almacenadas en AlertControl y *kill* elimina el smart contract.

```
1  function newAlert(bytes32 info) payable public ;
2  function verifyAlert(address a) payable public returns(bytes32);
3  function getPendingList() view public returns(address);
4  function getValue(address a)view public returns(bytes32);
5  function kill() payable  public ;
```

Las alertas se representan como smart contracts cuyos atributos aparecen en el siguiente código.

```
1  address _owner;
2  address private _creator;
3  uint256 private _price;
4  bytes32 private _signature;
```

Sus funciones son los *getters* de los atributos y la función *kill,* que elimina una alerta. Se pueden ver en el siguiente código

```
function getCreator() view public returns(address)
function getPrice() view public returns(uint256)
function getSignature() view public returns(bytes32)
function kill() public
```

4.6.4.Server

Se trata de la pieza central del prototipo e implementa todas las funciones que aparecen en el punto 4.4.1. Una vez creado este contrato se registra como *owner* de *Database* y *AlertControl* de tal manera que cuando es eliminado se encarga de borrar todos los smart contracts que forman el prototipo.

Este smart contract crea y almacena los rescates. Por otro lado es el enlace entre los usuarios y *AlertControl*.

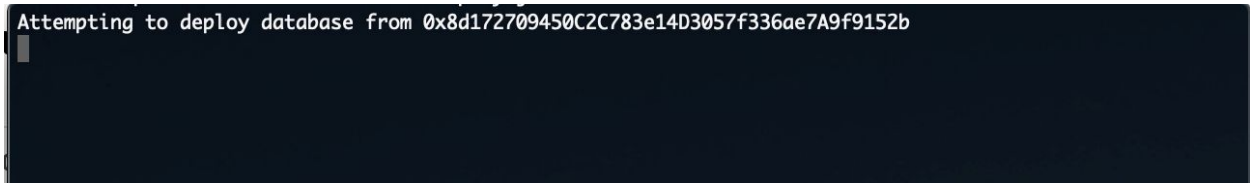
En el siguiente código se observan las variables más relevantes de *Server*. *Owner* representa al *address* que creó el smart contract Server y por lo tanto tiene la capacidad de eliminarlo. *currentRescues* almacena los rescates que aún no han terminado y, para evitar costes de gas extra, se utiliza el diccionario *registeredRescues* que dada la descripción de un rescate indica si pertenece a *currentRescues* o no.

```
address private Owner;
Accident [] currentRescues;
address dataBase;
address alertManager;
mapping(bytes32 => bool)private registeredRescues;
```

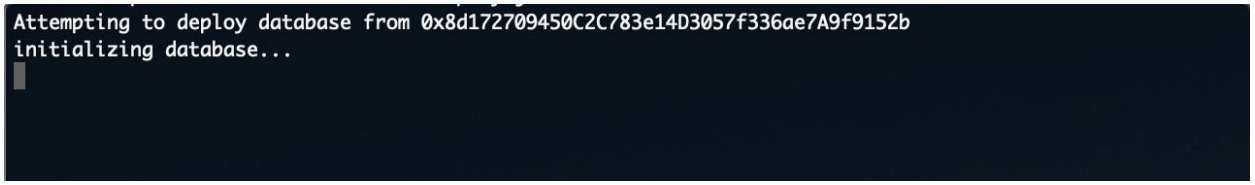
4.6.4.Desplegando el prototipo en Rinkeby.

Para comprobar que el prototipo puede ser desplegado en Rinkeby sin problemas se ha diseñado un programa JavaScript que utiliza Web3 para interactuar con el blockchain de Rinkeby (su código se encuentra en [55]), al cual se accede a través de nodos proporcionados por Infura. Para realizar la interacción se han usado cuentas de Metamask.

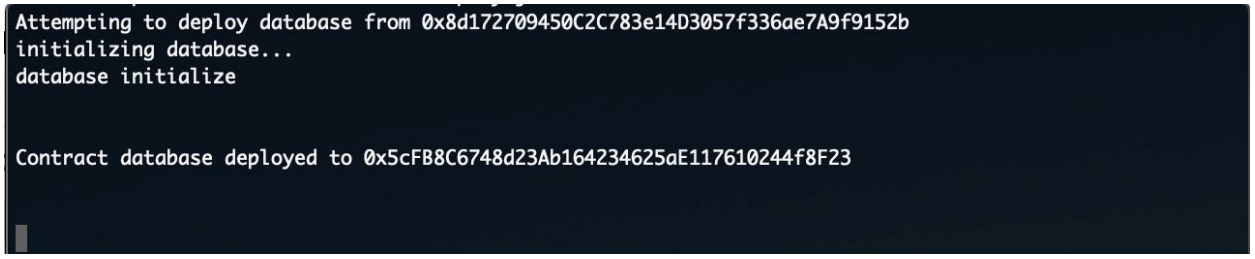
En la siguiente imagen se observa que el programa esta intentando desplegar el smart contract *Database* desde la cuenta *0x8d1727…..152b* en Rinkeby



Cuando el smart contract *Database* se ha desplegado en Rinkeby comienza su inicialización. En este momento el programa JavaScript esta cogiendo los datos de un fichero local y enviándolos al smart contract *Database* mediante transacciones en el blockchain de Rinkeby.



Durante este proceso de inicialización se observa una de las características más importantes de la interacción con un blockchain. El programa JavaScript ha introducido a 11 usuarios ficticios en *Database*, esto le ha llevado casi dos minutos. Aún siendo un blockchain de prueba el tiempo de espera por transacción es de unos 10 segundos (15 segundos en el blockchain principal de Ethereum). Cuando la base de datos se ha inicializado el programa nos indica la dirección del smart contract en Rinkeby.



Para comprobar que esto es así solo es necesario ir al etherscan de Rinkeby [48] e introducir el *address* que aparece en la imagen y que se ha dejado escrito en una tabla a final de este apartado para que al lector le sea sencillo introducirla en [48]. En Rinkeby se observa el estado del contrato *Database* como se ve en la siguiente imagen. En ella se observan las 11 transacciones para introducir los usuarios del prototipo y la transacción de creación, todas ellas enviadas por el mismo address que desplegó el contrato en Rinkeby.

Txn Hash	Block	Age	From		To	Value	[Txn Fee]
0xd158d5a96b277c...	4467084	2 hrs 54 mins ago	0x8d172709450c2c...	IN	0x5cfb8c6748d23a...	0 Ether	0.000028932
0x899f62ee71447e0...	4467078	2 hrs 56 mins ago	0x8d172709450c2c...	IN	0x5cfb8c6748d23a...	0 Ether	0.000085986
0x3f1c33e5afebab8...	4467077	2 hrs 56 mins ago	0x8d172709450c2c...	IN	0x5cfb8c6748d23a...	0 Ether	0.000086754
0x9e293d88a97253f...	4467076	2 hrs 56 mins ago	0x8d172709450c2c...	IN	0x5cfb8c6748d23a...	0 Ether	0.000070538
0xc31d6dae08b122...	4467075	2 hrs 57 mins ago	0x8d172709450c2c...	IN	0x5cfb8c6748d23a...	0 Ether	0.000070602
0x893ec51df8970a5...	4467074	2 hrs 57 mins ago	0x8d172709450c2c...	IN	0x5cfb8c6748d23a...	0 Ether	0.000070602
0x2afc55203dde5b...	4467073	2 hrs 57 mins ago	0x8d172709450c2c...	IN	0x5cfb8c6748d23a...	0 Ether	0.000070858
0x0820f77d7b68eb...	4467072	2 hrs 57 mins ago	0x8d172709450c2c...	IN	0x5cfb8c6748d23a...	0 Ether	0.000070858
0x09102b2e6b3738...	4467071	2 hrs 58 mins ago	0x8d172709450c2c...	IN	0x5cfb8c6748d23a...	0 Ether	0.000070602
0xa18fdfbd2c59bbc...	4467070	2 hrs 58 mins ago	0x8d172709450c2c...	IN	0x5cfb8c6748d23a...	0 Ether	0.000086242
0x11e3736b10a5b8...	4467069	2 hrs 58 mins ago	0x8d172709450c2c...	IN	0x5cfb8c6748d23a...	0 Ether	0.000070666
0x426df66525198d...	4467068	2 hrs 58 mins ago	0x8d172709450c2c...	IN	Contract Creation	0 Ether	0.000414144

Continuando con el despliegue de Galactic Chain en Rinkeby, como se puede comprobar en la siguiente imagen se han desplegado los smart contracts *ContractList* y *AlertControl* (todas las direcciones se dejarán al final de la sección para que el lector si lo desea pueda introducirlas fácilmente en [48]). Cabe destacar la última línea, lo que ha ocurrido es que desde la cuenta con la que se desplegó *ContractList* (*0x8d1727.....152b*) se ha enviado una transacción que llama a la función *declareOwnership* de *ContractList* para así declarar a *AlertControl* como nuevo *owner* de *ContractList*, ya que es quien va a utilizar dicho smart contract.

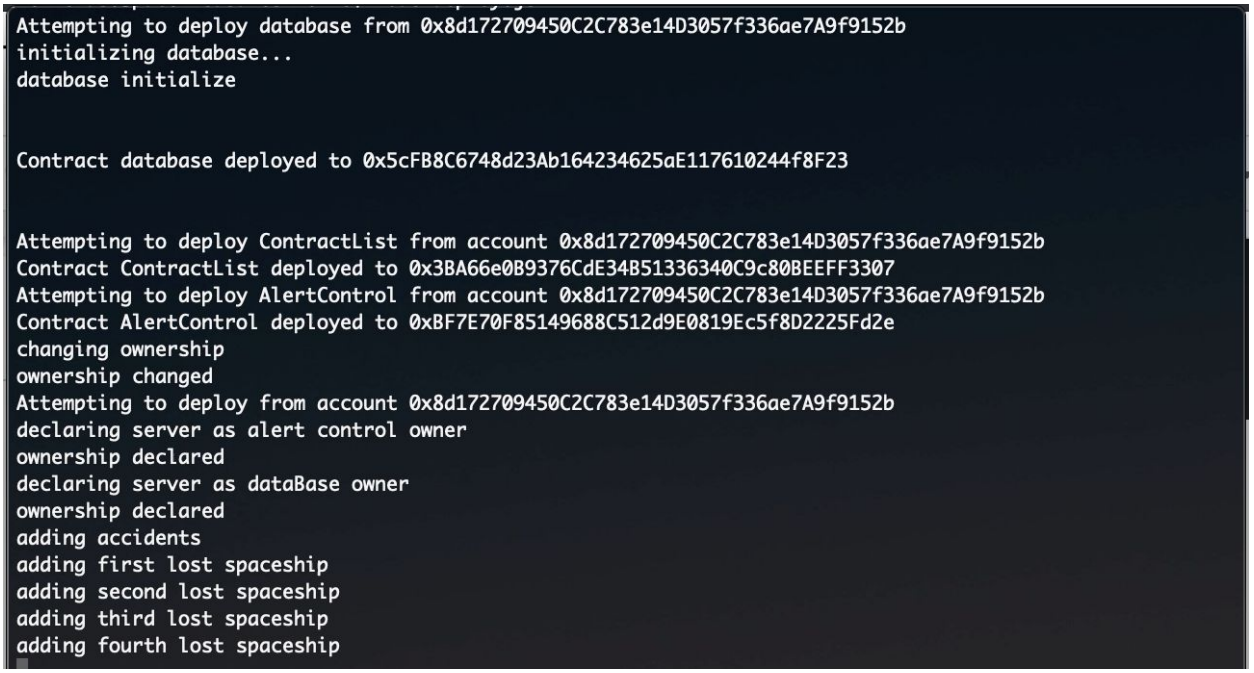
```
Attempting to deploy database from 0x8d172709450C2C783e14D3057f336ae7A9f9152b
initializing database...
database initialize


Contract database deployed to 0x5cFB8C6748d23Ab164234625aE117610244f8F23


Attempting to deploy ContractList from account 0x8d172709450C2C783e14D3057f336ae7A9f9152b
Contract ContractList deployed to 0x3BA66e0B9376CdE34B51336340C9c80BEEFF3307
Attempting to deploy AlertControl from account 0x8d172709450C2C783e14D3057f336ae7A9f9152b
Contract AlertControl deployed to 0xBF7E70F85149688C512d9E0819Ec5f8D2225Fd2e
changing ownership
```

Avanzando hasta el final de la ejecución se observa el terminal como se ve en la siguiente imagen. *Server* se ha registrado como *owner* de *Database* y de Alert*Control*, pudiendo así borrar todo el prototipo del blockchain con solo llamar a la función *kill* de *Server*. Además

se han incluido cuatro alertas de naves desaparecidas en el prototipo, desde [48] puede verse cuanto *ether* se ha mandado como recompensa por cada alerta.



Con esto concluye el proceso de desplegar Galactic Chain en un blockchain, la ejecución ha tardado un total de 6 minutos. En la siguiente tabla se pueden ver todos los address por si el lector quiere comprobar las transacciones en [48]

Address desde la que se han creado los smart contracts	0x8d172709450C2C783e14D3057f336ae7A9f9152b
Server	0x3d1AE5313a111e26D9F88dfa43aab6143fb5E065
ContractList	0x3BA66e0B9376CdE34B51336340C9c80BEEFF3307
AlertControl	0xBF7E70F85149688C512d9E0819Ec5f8D2225Fd2e

Conclusiones de la parte 2

En esta parte se ha explicado la infraestructura habitual que utilizan los desarrolladores de smart contracts para comunicarse con Ethereum. Al final de esta parte el lector no debería ver a Ethereum como un ecosistema cerrado e impenetrable sino como un ecosistema con el que se puede interactuar.

La infraestructura aquí explicada no es la única que permite interactuar con Ethereum, existen muchas otras. En esta parte se ha decidido utilizar esta infraestructura por lo sencillo que es montarla. Dado que este capítulo esta centrado en los desarrolladores de aplicaciones descentralizadas, esto es una cualidad fundamental.

Todos los servicios que forman esta infraestructura han sido utilizados para desarrollar la aplicación descentralizada Galactic Chain. Es innegable que todos ellos han facilitado mucho el desarrollo, sin embargo no se debe olvidar que son servicios que el desarrollador no puede controlar y que pueden surgir incompatibilidades.

Por otro lado todos servicios utilizados son bastante recientes, al igual que la tecnología blockchain. Esto provoca que el periodo de tiempo entre versiones sea muy corto, que sumado a las incompatibilidades entre servicios, hace que mantener actualizada una aplicación descentralizada requiera un esfuerzo constante.

Por último, en esta parte se ha presentado la aplicación descentralizada Galactic Chain. Aunque a primera vista esta aplicación parece ser una caso de uso válido de aplicación descentralizada, este caso de uso incumple varios de los principios aprendidos en la parte 1.

Galactic Chain necesita almacenar una cantidad demasiado alta de información para poder funcionar correctamente. Si a esto se le suma que la aplicación está destinada a utilizarse por un gran número de individuos, llegaría un momento en el que utilizar la aplicación sería imposible. Por otro lado, almacenar una gran cantidad de información provoca que algunas de las operaciones que más utiliza Galactic Chain sean muy costosas y necesite más de una transacción para completarlas.

Independientemente de que Galactic Chain esté diseñada para ser un intermediario entre los equipos de rescate y los dueños de las naves, la gran cantidad de información que tiene que almacenar y el coste elevado de algunas de sus operaciones hacen que no pueda ser una aplicación descentralizada. En Galactic Chain se observa que una de las características fundamentales de las aplicaciones descentralizadas es que tienen que ser escalables, si no, están abocadas al fracaso.

Por otro lado, cada vez son más las nuevas técnicas que se investigan para dotar a la tecnología blockchain de una capacidad de cómputo mayor. Entre ellas parece muy prometedor el uso de *sidechains* [42].

Un *sidechain* es un blockchain generado a partir de una bifurcación del blockchain principal. Esta bifurcación no afecta al blockchain principal y ambas cadenas pueden volver a unirse en un futuro, lo que lo hace distinto de un *hard fork* [42].

La necesidad de incluir *sidechains* en los blockchains surge a raíz de la gran cantidad de trabajo que el blockchain principal tiene que hacer. Los *sidechains* representan una plataforma especializada capaz de realizar una tarea concreta de forma rápida.

Gracias a esta técnica la escalabilidad del blockchain puede mejorar considerablemente haciendo que máquinas virtuales como la de Ethereum tengan mayor potencia de cómputo, lo que afectaría considerablemente a las viabilidad de Galactic Chain.

Parte 3

Introducción a la parte 3

En la parte 1 se concluye que la utilidad de las aplicaciones descentralizadas es la de servir de intermediarias en procesos simples que no requieren mucha memoria ni cómputos costosos.

Las aplicaciones descentralizadas pueden servir de intermediario porque su funcionamiento lo determina el código que las forma. Dado que el código toma las decisiones, para confiar en una aplicación descentralizada como intermediaria, un usuario debe confiar en que su código no contenga ni fallos ni trampas.

El código de un smart contract almacenado en el blockchain, y que puede obtenerse, no es de gran utilidad ya que se trata de código de byte, que es ilegible por un humano. Esta parte se va a centrar en el código Solidity de los smart contracts. Como conclusión a esta parte se espera determinar si, aun teniendo el código de una aplicación descentralizada en un lenguaje de alto nivel, un usuario tiene fácil determinar si ese código hace lo que dice.

5.¿Con qué tipo de ataques y problemas se pueden encontrar los smart contracts en Ethereum?

Situaciones a tener en cuenta durante el diseño de smart contracts.

Normalmente una aplicación se visualiza como dos componentes. Por un lado los datos que la aplicación utiliza, almacena y modifica. Por otro lado el código que interactúa con los datos de la aplicación. Cuando se desarrolla una aplicación los datos son independientes del código en el sentido que se puede acceder a ellos sin necesidad de tener que recurrir al código, siempre que se tenga acceso al lugar en el que están almacenados.

Esta división de una aplicación en código y datos tiene otra ventaja importante, la posibilidad de actualizar el código sin perder los datos. Si se quiere hacer una nueva versión del código basta con desarrollarla y suministrarle los datos de la versión anterior.

Una de las características más importantes de los smart contracts es que, entre otras cosas, representan la unión del código y los datos que utiliza. Ambos elementos son inseparables, los datos no pueden accederse si no es a través del código y el código sólo tiene sentido si utiliza dichos datos.

A esta característica se le suma el hecho de que el código de un smart contract es inmutable y por lo tanto crear una nueva versión del código implica crear un nuevo smart contract que almacene esta versión. Además, si la nueva versión debe utilizar los datos de la antigua, el antiguo smart contract debe ser capaz de transferir sus datos al nuevo smart contrac lo cual puede llegar a ser algo muy costoso.

Todas estas peculiaridades hacen que a la hora de diseñar un smart contract no solo se deban incluir mecanismos para transferir los datos entre versiones, también se deben diseñar con la intención de tener que crear el menor número de versiones posibles.

Para que un smart contract no tenga que ser actualizado con regularidad es importante conocer de antemano las vulnerabilidades que existen y las soluciones que se proponen.

A lo largo de este capítulo se expondrán algunos de los errores más conocidos a la hora de programar smart contracts, así como otros no tan conocidos. Se expondrán las soluciones que se han utilizado en el diseño de Galactic Chain. Por otro lado se expondrá cómo Galactic Chain permite el paso de datos entre versiones. El código que se utilizará en los siguientes apartados forma parte de Galactic Chain en algunos casos. Tanto el código que forma parte de Galactic Chain como el que no se encuentra en [55].

5.1.A todo gas

Como ya se explicó en el capítulo 3, una de las peculiaridades de los smart contracts en Ethereum es que la ejecución de sus funciones está limitada por la cantidad de gas que pueden

consumir. El gas en Ethereum es un recurso valioso no porque cueste ether, ya que se trata de una cantidad mínima de ether, sino porque limita las acciones que podrá realizar un smart contract.

Pero, ¿dónde está el problema? Al fin y al cabo, si la ejecución ha fallado por falta de gas solo se debe repetir dicha ejecución procurando enviar más gas. Si bien esto puede solucionar la situación en determinadas situaciones, en otras no es así.

La cantidad de gas que puede utilizar un smart contract no la limita solo quien lanza la ejecución. Todas las ejecuciones se realizan dentro del bloque que se está minando y que cuenta con un límite de gas por transacción. Por lo tanto, la ejecución de una función puede costar tanto gas que exceda el límite del bloque. La única manera de salir de una situación así es esperar a que la ejecución se encuentre dentro de un bloque con un límite de gas lo suficientemente alto como para completarla, sin embargo esta situación puede no darse nunca.

Se considera que un smart contract tiene vulnerabilidades por falta de gas cuando puede llegar a un estado en el que hay un subconjunto no vacío de sus funciones que no pueden volver a ser ejecutadas, pues la ejecución siempre terminará con una excepción por falta de gas.

A continuación se van a detallar y ejemplificar tres situaciones en las que un mal diseño puede llevar a un smart contract a situaciones de gas insuficiente. Algunas de estas situaciones le serán familiares a aquellos que hayan programado en otros lenguajes Turing completos. Sin embargo también se observarán situaciones que muestran peculiaridades de Ethereum que no pueden pasar desapercibidas a los diseñadores de smart contracts.

5.1.1.Operaciones sobre conjuntos no acotados

Dado el ejemplo de código Java mostrado en la figura 30, ningún programador pensaría que un objeto de esa clase podría llegar a un estado irreversible en el cual la operación *massOperation()* no se pueda ejecutar. Se trata de una operación sobre un conjunto de elementos que, a pesar de poder ser todo lo grande que le permita la memoria, siempre será un conjunto finito y por lo tanto *massOperation* siempre terminará.

Puede comprobarse que el smart contract de la figura 31 representa la misma lógica que la clase Java de la figura 30. Sin embargo en este caso, debido al entorno en el que se encuentran los smart contracts, el smart contract *Unsafe* es vulnerable a quedar en un estado en el cual la operación *massOperation* no pueda ejecutarse más.

Aunque el array *a[]* seguirá teniendo un número de elementos finito en cualquier estado, la operación *massOperation()* corre el riesgo de que en algún momento *a[]* sea tan grande que la operación no pueda ejecutarse sin provocar una excepción por falta de gas. El gas necesario podría ser mayor que el gas permitido por el bloque actual y que el de varios bloques futuros, dejando el contrato bloqueado durante un periodo largo de tiempo.

De este modo un atacante que desee dejar al smart contract *Unsafe* en un estado en el que *massOperation()* solo tendrá que llamar a la operación *addInt()* un número suficientemente grande de veces [32].

```
1   class SafeClass
2   {
3           private ArrayList<int> a;
4           public SafeClass()
5           {
6                this.a = new ArrayList<int>();
7           }
8           public void addInt(int n)
9           {
10              this.a.add(n);
11          }
12          public void massOperation()
13          {
14               for(int i = 0; i < this.a.size();i++)
15                    this.a.set(i,this.a.get(i)+1);
16          }
17  }
```

Figura 30: clase Java

```
1   contract Unsafe
2   {
3           uint[] private a;
4           function addInt(uint n) public
5           {
6                a.push(n);
7           }
8           function massOperation() public payable
9           {
10              uint i = 0;
11              while(i < a.length && msg.gas > 100000)
12              {
13                              a[i] = a[i]+1;
14                               i++;
15              }
16          }
17  }
```

Figura 31: smart contract equivalente al código de la figura 30 pero vulnerable

Los smart contracts son inmutables y por lo tanto una vez que *Unsafe* esté en Ethereum ya no habrá ningún modo de protegerlo. Una posible solución para que *Unsafe* quede protegido una vez sea subido a Ethereum sería diseñar la operación *massOperation()* como indica la figura 32 [32].

```
1   contract Unsafe
2   {
3         uint[] private a;
4         uint nextAccount;
5         function Unsafe() public
6         {
7               nextAccount = 0;
8         }
9         function addInt(uint n) public
10        {
11              a.push(n);
12        }
13        function massOperation() public payable
14        {
15              uint i = nextAccount;
16              while(i < a.length && msg.gas > 100000)
17              {
18                            a[i] = a[i]+1;
19                           i++;
20              }
21               nextAccount = i%a.length;
22        }
23  }
```

Figura 32: smart contract Unsafe sin vulnerabilidades

Ahora, el smart contract *Unsafe* tiene en cuenta en su operación *massOperation()* que la ejecución de instrucciones cuesta gas y que puede ser que en una misma ejecución no se puede completar la operación. Por ello guarda dentro de su estado, gracias a *nextAccount*, por donde debe continuar la ejecución.

Operaciones sobre conjuntos no acotados en Galactic Chain:

En el prototipo de Galactic Chain existe una operación que opera sobre un conjunto no acotado. Esta función, *rewardAccident*, se encarga de eliminar del sistema un rescate que ya ha terminado. Debido a que la función consume mucho gas puede necesitar más de una transacción para completarse.

Cuando la ejecución de eliminar un rescate necesita más de una transacción para completarse, no se puede permitir que otro tipo de transacciones que necesitan la información de los rescates actualizada se ejecuten. Para evitar esto el smart contract entra en un estado especial en el que, mientras los rescates no estén actualizados, todas las funciones que necesitan dicha información actualizada fallan.

El objetivo es que todos aquellos usuarios que quieren utilizar funciones que necesitan los rescates actualizados ayuden a terminar antes el trabajo de actualizar.

Esto provoca que un usuario de Galactic Chain, antes de utilizar una función, debe comprobar si queda trabajo de actualizar por hacer, como expresa la figura 33.

```
1  smartContract GalacticChain;
2  while( GalacticChain.homework())
3        GalacticChain.rewardAccident();
4  //otras llamadas que utilizan la información de los rescates
```

Figura 33: proceso a seguir cuando se realiza una llamada a Galactic Chain que utilizará información de los rescates.

Las transacciones que se encuentran a partir de la línea 4 darán error si queda trabajo de actualizar por hacer. Por lo tanto, antes de enviarlas, el usuario comprueba si queda trabajo por hacer. En caso afirmativo el usuario ayuda a completar dicho trabajo llamando a la función de la línea 3, que es la que se encarga de la actualización de los rescates.

5.1.2.Llamadas no aisladas a funciones externas

Si la figura 34 pertenece a un smart contract entonces cualquiera que haya conseguido introducir otro smart contract como inversor en el array *investors* podría provocar que la función en la que se encuentra este código nunca se ejecute sin producir una excepción por falta de gas. Todo lo que tiene que hacer el atacante es declarar en un smart contract una función fallback que consume más gas del permitido (2300 gas en Solidity) e introducir ese smart contract como inversor. Desde ese momento cada vez que el bucle ejecute la operación send para dicho smart contract se provocará dicha excepción.

```
1  for (uint i = 0;  i < investors.length; i++)
2  {
3        if (investors[i].invested < min_investment)
4        {
5              if( !(investors[i].addr.send(investors[i].dividendAmount))) throw;
6              investors[i] = newInvestor;
7        }
8  }
```

Figura 34: bucle con llamadas a funciones externas.

Una forma muy similar a la ya mencionada anteriormente de eliminar esta vulnerabilidad se puede ver en la figura 35 [32].

```
1   contract ...{
2   uint nextAccount = 0;
3   function  ...()...{
4         for (uint i = 0;  i < investors.length; i++)
5         {
6               if (investors[i].invested < min_investment)
7               {
8                                 nextAccount = i+1 % investors.length;
9                     if(  !(investors[i].addr.send(investors[i].dividendAmount)))
10                                                throw;
11                    investors[i] = newInvestor;
12              }
13        }
14  }
```

Figura 35: solución al bloqueo por culpa de llamadas a funciones externas no aisladas.

Este el smart contract guarda en storage la variable *nextAccount*. Esta variable indica por dónde tiene que continuar el bucle la próxima vez que se ejecute la función en la que reside y por lo tanto no quedará bloqueada por aquellas cuentas que provocan fallos al recibir fondos.

En Galactic Chain se tuvo que considerar esta vulnerabilidad durante el tratamiento de alertas. Para entender por qué aparece esta vulnerabilidad antes hay que entender la manera de afrontar el tratamiento de alertas de Galactic Chain.

Alertas en Galactic Chain:

Galactic Chain tiene definida la función de la figura 36, que se encarga de agrupar un conjunto de alertas dentro de un array y enviar este a través de un smart contract de tipo *LostList*.

Se puede observar que tanto en la línea 6 como en la 8 hay llamadas a funciones externas, en concreto a funciones que pertenecen al address *act* declarado en la línea 3. Sin embargo esta función no hace vulnerable al sistema.

Las alertas en Galactic Chain tienen un periodo de vida muy corto,pues se espera que la creación y eliminación de alertas sean operaciones frecuentes. Además el número de alertas que puede llegar a controlar Galactic Chain puede ser elevado. Debido a esto el sistema debe almacenarlas en alguna estructura de datos que permita manipularla con unos mecanismos de inserción y eliminación rápidos. Esta estructura de datos es un smart contract que funciona como una lista doblemente enlazada de *addresses*. Está representada como se indica en la figura 37.

Al tomar la decisión de implementar la estructura de datos como un smart contract, hay que tener en cuenta que cualquier cuenta de Ethereum podría interactuar con la estructura. El atributo de la línea 4 de la figura 37 permite comprobar si una operación sobre la lista la ha

iniciado Galactic Chain o una cuenta posiblemente maliciosa. Las acciones de insertar y eliminar en la lista pueden verse en la figura 38. Se puede observar que ambas funciones consumen poco gas y carecen de bucles, lo que hace de esta una estructura ideal para almacenar alertas.

```
1   function getPendingList() payable public returns(address)
2   {
3               address act = ContractList(list).top();
4               address [] memory ret = new address[](length);
5               uint256 i = 0;
6         while(ContractList(list).hasKey(act) && msg.gas > 10000000)
7         {
8                       if(Node(act).signedByMe()){
9                             ret[i] = act;
10                            i++;
11                      }
12                      act = ContractList(list).getNext(act);
13              }
14              address a = new LostList(ret,msg.sender);
15            return a;
16  }
```

Figura 36: función de Galactic Chain que devuelve un subconjunto de las alertas guardadas en la lista de alertas.

```
1   Contract List{
2         address private ini;
3         address private end;
4         address private owner;
5         mapping(address => bool) private isInTheStack;
6         mapping(address => address) private nodeTonext;
7         mapping(address => address)  private nodeToprev;
8         .
9         .
10        .
```

Figura 37: atributos que representan una lista enlazada de address.

Las llamadas a funciones externas de la figura 36 son seguras debido a que el único smart contract que puede incluir elementos dentro de *List* es aquel cuyo *address* sea igual a su atributo *owner* y por lo tanto no hay *address* desconocidos que pudiesen atacar a este smart contract. Por lo tanto es imposible que las llamadas de las líneas 6 y 8 de la figura 36 ejecuten una *función de fallback*.

Por otro lado este bucle no genera una vulnerabilidad como la expuesta en el apartado 5.1.1 ya que las alertas se procesan en conjuntos. El bucle tiene en cuenta el gas que le queda y devolverá un conjunto de alertas tan grande como el gas le permita. Cuando estas alertas se

procesen, la función del código de la figura 36 podrá devolver un conjunto nuevo ya que las del conjunto anterior se han eliminado.

```
1  function erase(address n) OnlyOwner payable public
2  {
3        require(isInTheStack[n]);
4              if(ini == n)
5              {
6                    ini = nodeTonext[ini];
7                    nodeToprev[ini] = address(0);
8              }
9              else{
10                   address aux = nodeToprev[n];
11                   nodeTonext[aux] = nodeTonext[n];
12                   nodeToprev[nodeTonext[n]] = aux;
13             }
14             isInTheStack[n] = false;
15  }
16
17 function push(address n) OnlyOwner payable public
18 {
19       isInTheStack[n] = true;
20             nodeTonext[n] = ini;
21             nodeToprev[n] = address(0);
22             nodeToprev[ini] = n;
23             ini = n;
24  }
```

Figura 38: operaciones de insertar y eliminar en la lista enlazada de smart contracts.

5.1.3.Overflow de enteros

Si bien hasta el momento los problemas que se han presentado se pueden achacar a la falta de cuidado por parte del programador, que olvida tener en cuenta el entorno para el que está programando, en este caso se expone un fallo de base en el compilador de Solidity a bytecode y que los programadores de smart contracts deben tener en cuenta para evitar vulnerabilidades en su código [32].

En el caso del smart contract de la figura 39 un atacante no tendría que preocuparse de cuántos elementos debe introducir en data[] para que el bucle genere una excepción por falta de gas. Desafortunadamente, en este caso al atacante le bastaría con introducir 256 elementos. Esto se debe a que Solidity convierte el entero i de tipo *var* a un *uint8*, mientras que los índices de un array están acotados por enteros de tipo *uint256*. Esto quiere decir que si *data.length* es mayor de 256 entonces i sufrirá un overflow, ya que, en aritmética de 8 bit, 255 + 1 = 0. El bucle volverá a empezar desde el principio y en algún momento provocará una excepción por falta de gas.

```
1  contract Overflow {
2        Payee data[];
3        function goOverAll() {
4                    for (var i = 0; i < data.length; i++) { ... }
5        }
6        ...
7  }
```

Figura 39: contrato diseñado sin utilizar el sistema de tipos de Solidity.

```
1   uint256 stateMiss;
2   function deleteAccidentProtocol() private
3   {
4     while(stateMiss < currentAccidents[currIdAC].missing.length && msg.gas > 1000)
5     {
6           registerExisting[currentAccidents[currIdAC].missing[stateMiss]]= false;
7           stateMiss++;
8     }
9         …..
10  }
```

Figura 40: función diseñada teniendo en cuenta los tipos de las variables

Overflow de enteros en Galactic Chain:

Durante el desarrollo de Galactic Chain se han evitado estos fallos procurando no utilizar Solidity como un lenguaje no tipado. En el diseño de smart contracts es crucial pensar el tipo de una variable si se quieren evitar situaciones como las que produce la figura 39.

En el desarrollo de un bucle de Galactic Chain como la figura 40 se ha tenido en cuenta que la variable *stateMiss* va a compararse con la longitud de un array que es de tipo *uint256* y por lo tanto *stateMiss* también se ha declarado como *uint256*.

Con esta vulnerabilidad concluyen los casos en los que un smart contract puede incluir vulnerabilidades por falta de gas que se expondrán en este texto. Si se puede sacar una conclusión de estos casos es la de que este tipo de vulnerabilidades no pueden tomarse a la ligera ya que pueden tener consecuencias muy graves sobre un contrato.

5.2.¿Yo no he pasado ya por aquí?, El problema de la reentrada

El invariante de un smart contract es aquel predicado que sus atributos deben satisfacer cuando no se está ejecutando ninguna de sus funciones. Las funciones de un smart contract se ejecutarán como se espera si antes de su ejecución se cumplía el invariante.

La vulnerabilidad de reentrada se alimenta de un fallo muy común en programación. La construcción de estructuras en las que eventualmente los atributos no satisfacen su invariante y no hay ningún fragmento de su código en ejecución. A lo largo de este apartado a este fallo se le llamará *fallo de invariante* [36].

```
1   contract Count
2   {
3         uint256 total;
4         uint256 [] nums;
5         function Count() public
6         {
7               total = 0;
8         }
9
10        function getTotal() returns(uint256)
11        {
12              return total;
13        }
14        function add(uint256 a )
15        {
16              nums.push(a);
17        }
18
19  }
```

Figura 41: smart contract con fallo de invariante.

En la figura 41 se implementa el smart contract *Count*. *Count* posee dos atributos, un array de elementos uint256 y la suma de todos los elementos de dicho array. El invariante de Count puede expresarse como $\{ total = \sum i: 0 \leq i \leq nums.length : nums[i] \}$.

Cuando se crea una instancia de *Count*, sus atributos cumplen el invariante, pero tras ejecutar la función *add* se añade un elemento a *nums* y no se suma su valor a *total.* Esto genera un *fallo de invariante* que hará que la función *getTotal* no vuelva a funcionar correctamente.

La versión correcta de cómo implementarlo se muestra en la figura 42.

Percatarse del *fallo de invariante* de la figura 41 es bastante sencillo. Sin embargo, en funciones complejas, con llamadas a otras funciones dentro de su código, comprobar si antes de ejecutar cada llamada se cumple el invariante puede ser muy complicado.

```
1   contract Count
2   {
3         uint256 total;
4         uint256 [] nums;
5         function Count() public
6         {
7               total = 0;
8         }
9
10        function getTotal() returns(uint256)
11        {
12              return total;
13        }
14        function add(uint256 a )
15        {
16              nums.push(a);
17              total += a;
18        }
19
20  }
```

Figura 42: smart contract Count sin fallo de invariante.

Además, el desarrollo de smart contracts tiene una dificultad añadida y es que el código de las funciones externas que se utilicen puede no estar disponible en tiempo de compilación. Esto implica que cualquier llamada a una función externa puede generar reentradas. Si esto ocurre en un smart contract con *fallo de invariante* el correcto funcionamiento de este puede verse comprometido.

La figura 43 es un ejemplo más complejo de cómo un smart contract puede ser vulnerable ante una reentrada por culpa de un *fallo de invariante*. En este código *Safe* simula el funcionamiento de una caja fuerte compartida de *ether*. Un address puede guardar *ether* dentro de este smart contract así como ir cogiendo el dinero que tiene guardado.

Por otro lado, en la figura 44 se presenta el smart contract *Investor* que representa a un usuario de *Safe*. Un Investor posee una caja fuerte y controla la cantidad de *ether* que puede guardar y sacar de la caja fuerte a voluntad.

La línea 17 de esta figura muestra un tipo de sintaxis habitual en Solidity pero que puede ser nueva para el lector. La instrucción *Investor(msg.sender)* devuelve un address, al concatenar *.addMoney* se indica la función que se desea llamar y, por último, al concatenar *.value(money)()* se indica que se va a enviar en dicha llamada una cantidad de *ether* igual a *money*.

```
1   contract Safe
2   {
3         mapping(address => uint) balance;
4         function Safe()payable public
5         {
6               addOwner();
7         }
8
9         function addOwner() payable public
10        {
11              balance[msg.sender] += msg.value;
12        }
13
14        function getMoney(uint money) payable public
15        {
16              if(money <= balance[msg.sender]){
17                    Investor(msg.sender).addMoney.value(money)();
18                    balance[msg.sender] -= money;
19              }
20        }
21  }
```

Código 43: smart contract vulnerable ante una reentrada por culpa de un fallo de invariante.

Ahora se explicará por qué una reentrada en la función *getMoney()* de *Safe* es perjudicial para el smart contract y por qué el *fallo de invariante* es el responsable de esto.

El tipo *address* en Ethereum puede representar a un smart contract, sin embargo no es posible saber a qué contrato referencia. Aunque la línea 17 de la figura 43 aparente realizar casting de tipos, no lo hace, se trata de azúcar sintáctico de la línea de ls figura 45.

Esto quiere decir que, en la instrucción de la línea 17 de la figura 43, el fragmento "*Investor(msg.sender)"* no incluye ningún tipo de comprobación en lo que al tipo del smart contract se refiere. La instrucción se limita a pasar un mensaje al *address* indicado, en caso de que este no sea un smart contract de tipo *Investor* se ejecutará la *función de fallback* del *address*.

```
1   contract Investor
2   {
3         uint balance;
4         Safe safe;
5         function Investor(address _safe) payable public
6         {
7               balance = msg.value;
8               safe = Safe(_safe);
9         }
```

```
10        function safeMoney(uint money) public
11        {
12              if(money <= balance){
13                    balance -= money;
14                    safe.addOwner.value(money)();
15              }
16        }
17        function takeBalance(uint money)public
18        {
19              safe.getMoney(money);
20        }
21        function addMoney() payable public
22        {
23               balance += msg.value;
24        }
25  }
```

Código 44: smart contract que interactuará con el smart contract Safe.

```
1   msg.sender.call.value(money)("addMoney");
```

Figura 45: instrucción "Investor(msg.sender).addMoney.value(money)()" sin azúcar sintáctico.

Un smart contract que explota la vulnerabilidad de reentrada en *Safe* es el de la figura 46. Este smart contract de 24 líneas tiene la capacidad de robar tanto dinero de *Safe* como desee. Un atacante sólo tendría que proceder del siguiente modo:

1. Crear un smart contract Attacker con el address de la caja fuerte que desea atacar.
2. Llamar a *safeMoney* con tanto dinero dinero como desee.
3. Llamar a la función *attack*

Esto es posible debido a que el diseñador del smart contract *Safe* no ha tenido en cuenta la reentrada en la función *Safe* y no deja los atributos de *Safe* en un estado en el que se satisfaga su invariante. Esto es lo que sucede cuando un atacante llama a la función *attack*:

1. La función *attack* llama a la función de *Safe getMoney* para que esta le devuelva el dinero que ha guardado en ella.
2. *Safe* en su función *getMoney* envía al *address* el dinero que había guardado en allí a través de la función *addMoney*.
3. El *address* que ha llamado a la función *getMoney* es un *address* del smart contract *Attacker* y carece de función *addMoney* por lo tanto se ejecutará la función *fallback* de *Attacker*.
4. Cuando se ejecuta la función *fallback* de *Attacker* esta vuelve a llamar a función *getMoney* de *Safe* si le queda gas y a *Safe* le quedan fondos.
5. Como *Safe* en la llamada anterior aún no le había restado los fondos correspondientes al address en cuestión (*fallo de invariante*), volverá a enviar la misma cantidad y por lo tanto quitando dinero de otros usuarios de *Safe*.

```
1   contract Attacker
2   {
3         uint balance;
4         Safe safe;
5         function Investor(address _safe) payable public
6         {
7               balance = msg.value;
8               safe = Safe(_safe);
9         }
10        function safeMoney() payable public
11        {
12              balance = msg.value;
13              safe.addOwner.value(msg.value)();
14        }
15         function attack() public
16        {
17              safe.getMoney(balance);
18        }
19        function ()external
20        {
21              if(msg.gas > 10000 && msg.sender.balance > balance)
22              msg.sender.call("getMoney",balance);
23        }
24  }
```

Figura 46: posible atacante de Safe

El invariante de *Safe* expresa que su balance debe ser igual al total de *Ether* que los inversores han almacenado. En la línea 17 de la figura 43 se está realizando una llamada a una función externa y por lo tanto, antes de que se ejecute, las variables de *Safe* deberían satisfacer su invariante. Esto no es así, la llamada a la función externa envía una cantidad de *ether* al destinatario, haciendo que el balance del smart contract disminuya. Sin embargo el registro de cuánto ha guardado cada *Investor* en *Safe* no se ha actualizado, provocando un *fallo de invariante* (esto se hace a continuación en la línea 18)

Para evitar este tipo de vulnerabilidades es importante conocer el invariante del smart contract que se está diseñando. Además el código debe asegurar que, al terminar una función o al realizar una llamada a una función externa, las variables del smart contract satisfacen su invariante.

```
1   contract Safer
2   {
3         mapping(address => uint) balance;
4         function Safe()payable public
5         {
6               addOwner();
7          }
8
9         function addOwner() payable public
10        {
11              balance[msg.sender] += msg.value;
12        }
13
14        function getMoney(uint money) payable public
15        {
16              if(money >= balance[msg.sender]){
17                    balance[msg.sender] -= money;
18                    Investor(msg.sender).addMoney.value(money)();
19              }
20
21         }
22  }
```

Figura 47: smart contract Safe sin fallo de invariante.

En la figura 47 se ha evitado el *fallo de invariante* de la figura 43. La línea 17 actualiza el balance del inversor antes de enviarle el dinero satisfaciendo así el invariante. En este caso, aunque se produjese una reentrada a *getMoney,* el atacante no podría llevarse más dinero del que invirtió.

Una vulnerabilidad de reentrada fue utilizada para atacar a The DAO, consiguiendo robar cerca de 3.600.000 *ether* [36][37], 50 millones de dólares en junio de 2016 [56].

Reentrada en Galactic Chain:

Uno de predicados que forman el invariante del contrato *Server* de Galactic Chain expresa que el balance que posee debe ser igual a la cantidad de Ether para recompensas que se le ha mandado.

La función de la figura 48 es la encargada de enviar al equipo ganador de un rescate la recompensa. Esto se hace a través de la instrucción de la línea 11. Como la ejecución de dicha instrucción puede ejecutar una *función fallback* externa antes de ejecutarla, las variables deben cumplir el invariante de Galactic Chain. Las líneas 8 y 9 se encargan de esto, guardando en una variable local el precio pagado y poniendo el registro de la recompensa a cero. De este modo se ha evitado un *fallo de invariante*.

```
1   function rewardAccident() payable public returns(bool)
2   {
3         require(currentAccidents[currIdAC].creator == msg.sender);
4         if(work){
5                     return false;
6         }
7
8         uint256 price = currentAccidents[currIdAC].price;
9         currentAccidents[currIdAC].price = 0;
10        if(currentAccidents[currIdAC].winner.send(currentAccidents[currIdAC].price)){
11                    bytes32 info = currentAccidents[currIdAC].info;
12                    registeredAccidents[info] = false;
13                    return true;
14        }
15        else{
16                    currentAccidents[currIdAC].price = price;
17        }
18  }
19
```

Figura 48: función que podría sufrir una reentrada en Galactic Chain

Aunque evitar *fallos de invariante* en los smart contracts es crucial para evitar vulnerabilidades como las que origina la reentrada, las consecuencias no serían tan graves si el código de las funciones externas se conociese de antemano.

Para asegurar esto Galactic Chain utiliza un smart contract de autenticación llamado *Database*. *Database* se encarga de almacenar los *address* que son de confianza, es decir, controlados por un usuario autorizado. Cuando se llama a una función de Galactic Chain, antes de comenzar su ejecución, *Database* debe confirmar que quien ha realizado la llamada está autorizado para ello.

Este proceso no elimina las vulnerabilidades que pueda tener Galactic Chain, pero, al reducir el número de *address* que pueden interactuar con el sistema, algunas de estas no podrán ser explotadas.

5.3.El orden importa

En este caso lo que se va a exponer no es tanto una vulnerabilidad, en el sentido de que el smart contract no se ve perjudicado por ello, si no una característica que de no tenerse en cuenta puede perjudicar a los usuarios del smart contract.

Para representar esta característica se va a utilizar el sistema construido para este TFG, Galactic Chain. La figura 49 muestra el fragmento de Galactic Chain encargado de firmar rescates y añadir ítems a las naves desaparecidas.

```
1   struct Accident
2   {
3          //.....
4         address creator;
5         address winner;
6         uint256 price;
7         bytes32 signature;
8         bool signed;
9         bytes32 info;
10        address [] missing;
11        //.....
12    }
13    Accident[] public currentAccidents;
14    //.....
15  function signRescue(uint256 idAc,bytes32 signature) ......
16  {
17         require(currentAccidents[idAc].assistance[msg.sender]);
18        require(currentAccidents[idAc].signature == signature);
19        require(!currentAccidents[idAc].signed);
20        currentAccidents[idAc].winner = msg.sender;
21        currentAccidents[idAc].signed = true;
22        SignedAccident(idAc);
23    }
24    //.....
25  function addMissing(uint256 idAc,address idAddress)......
26  {
27        require(currentAccidents[idAc].creator == msg.sender);
28        require(!currentAccidents[idAc].signed);
29        currentAccidents[idAc].missing.push(idAddress);
30        registerExisting[idAddress] = true;
31        NewLost(idAc);
32  }
```

Figura 49: fragmento de Galactic Chain que muestra las funciones usadas para firmar un rescate y añadir un ítem a una nave desaparecida.

Una de las acciones que permite este smart contract, mediante la función *signRescue,* es la de firmar un rescate. Un equipo de rescate, al encontrar la nave que estaba rescatando y comprobar que todos los ítems que deben estar en la nave lo están, o bien su ausencia está

justificada, llamará a esa función con la firma de la nave. De este modo queda registrado como legítimo rescatador de la nave y propietario de la recompensa.

Por otro lado la función *addMissing* permite al dueño de la nave desaparecida añadir nuevos objetos que se encontraban en la nave cuando despareció. Es importante observar las condiciones de la líneas 27 y 28 que restringen las llamadas a esta función sólo a los creadores del rescate y sólo a los accidentes que no estén ya firmados.

A simple vista, parece imposible que quien solicita un rescate pueda incluir en un accidente nuevos objetos sin que los equipos se den cuenta. Esto le daría al dueño de la nave una excusa para no pagar el rescate. Pero ¿es imposible de verdad?.

La respuesta es no, no es imposible, un equipo de rescate podría intentar firmar un rescate cuyos objetos tiene localizados y terminar firmando un rescate con objetos que no estaban en Galactic Chain cuando firmó.

Para entender cómo ocurre esto hay que tener claro cómo es la interacción de un usuario con un smart contract. Los usuarios emiten transacciones con las llamadas a las funciones del smart contract; estas transacciones se incluyen dentro de un bloque y se ejecutarán **en el orden en el que estén en el bloque** sobre el smart contract cuando el bloque es minado.

Por lo tanto, teniendo en cuenta esta característica, podría darse el siguiente caso:

1. El equipo de rescate encuentra la nave y comprueba que todos los ítems que se indican como desaparecidos en ella están localizados.
2. Como todos los objetos están en ella decide firmar el rescate, es decir, manda una transacción que cuando se ejecute llamará a la función *signRescue*.
3. Quien pidió el rescate se las ha apañado para mandar una transacción que ejecuta la función *addMissing* más o menos a la vez que el equipo de rescate manda su transacción. Esta transacción añadirá un ítem que nunca ha estado en la nave.
4. La transacción *addMissing* terminá ejecutándose antes que *signRescue*.
5. La transacción *signRescue* fue emitida para ser ejecutada sobre un rescate que tenía unos ítems y va a ejecutarse sobre el mismo rescate pero con un ítem más. Quién emitió la transacción no sabrá esto hasta que se haya minado el bloque y por lo tanto ya será demasiado tarde.
6. El equipo de rescate ha terminado firmando un rescate con un nuevo objeto que desconoce y por lo tanto exponiéndose a ser acusado de robo.

Podríamos pensar que la probabilidad de que las transacciones se procesen en este orden es baja. Pero un propietario de una nave podría sobornar a un minero para que fije el orden de las transacciones de forma que primero procesen las suyas.

Con este ejemplo se ha mostrado que los diseñadores de smart contract deben tener en cuenta que la transacción de un usuario puede ejecutarse en un estado distinto del que este observó y por lo tanto se deben incluir mecanismos que consigan que una transacción solo se ejecute si es sobre los valores que observó el usuario[36][37].

Una manera de conseguir esto es seguir la lógica de la instrucción *atomic swap* de la programación concurrente[36]:

atomic_swap(x, y, z) := si x = y => x = z

De esta manera las funciones que modifican variables del contrato deberían recibir los valores de esas variables que observó el usuario para así saber si la instrucción se debe ejecutar o no.

Una posible solución para la figura 49 es implementar la función *signRescue* como se expone en la figura 50.

```
1   function signRescue(uint256 idAc,bytes32 signature,bytes32 _hash)....
2   {
3         require(currentAccidents[idAc].assistance[msg.sender]);
4         require(currentAccidents[idAc].signature == signature);
5         require(!currentAccidents[idAc].signed);
6         require(keccak256(currentAccidents[idAc]) == _hash);
7         currentAccidents[idAc].winner = msg.sender;
8         currentAccidents[idAc].signed = true;
9         SignedAccident(idAc);
10    }
```

Figura 50: función *signRescue* que tiene en cuenta el estado que observó el equipo de rescate que la llama.

En esta versión se ha incluido el parámetro *_hash*, que es el hash del rescate tal y como lo observó quién llamó a la función. La línea 6 comprueba que el hash del rescate actual es igual al hash del rescate que observó el usuario al emitir la transacción, si no es así la llamada fallará. De esta forma se evita en este caso que un equipo de rescate firme un rescate que está en un estado distinto del que observó.

5.4.¿Errores nunca antes vistos?

Maurice Herlihy en su artículo "*Blockchains from a distributed computing perspective*" [36] observa la programación de smart contracts desde el punto de vista de la programación distribuida.

Este cambio en la forma de ver la programación de smart contracts consigue que aquellos fallos que parecen algo nunca visto antes revelen su similitud con fallos más que conocidos de la programación distribuida.

5.4.1.Smart contracts como monitores

En programación concurrente un monitor es una estructura con un cerrojo incorporado que es adquirido cuando se realiza una llamada a un método del monitor y liberado cuando el método termina. Además un monitor cuenta con las instrucciones *wait* y *signal*. *Wait* permite a

un proceso detener su ejecución en un punto y liberar el cerrojo. *Signal* permite que un proceso que fue paralizado por una instrucción *wait* obtenga el cerrojo y continúe su ejecución.

Para razonar sobre la corrección de un monitor se utiliza su invariante, un aserto que debe cumplirse siempre que no haya un proceso ejecutando alguna operación del monitor.

Si se observan los smart contracts como si de monitores se tratara el problema de la reentrada no parece tan extraño. Una llamada a una función externa sería como ejecutar una instrucción wait en un monitor ya que el smart contract pierde el control de la ejecución y por lo tanto el programador debe asegurarse de que antes de que esto ocurra el smart contract se encuentra en un estado correcto.

Si se trata el smart contract de la figura 43, Safe, como un monitor su invariante podría tener la siguiente forma:

$$this.balance = \sum : address\ a : balance[a]$$

Es decir, el balance del smart contract siempre tiene que ser igual a la suma del dinero que le han enviado otros *address*.

En la línea 17 de la figura 43 se está haciendo una llamada a una función de otro *address*, lo que equivaldría a un wait en un monitor. Sin embargo en el código de la figura 43 Safe queda en un estado en el que su invariante no se cumple ya que this.balance ha disminuido pero balance[a], donde 'a' es el receptor de la llamada, no, por lo que:

$$this.balance < \sum : address\ a : balance[a]$$

Efectivamente, como postula Maurice Herlihy, tomar Safe como un monitor ha mostrado la vulnerabilidad de reentrada y por lo tanto, de tratarse de un smart contract que va a ser subido a Ethereum, ha salvado a posibles inversores de perder su dinero.

Conclusiones de la parte 3

Tras comprobar el gran número de fallos de programación que se pueden cometer a la hora de desarrollar smart contracts, se puede concluir que programar aplicaciones descentralizadas no es algo sencillo. Es cierto que algunos de estos fallos ya se daban en escenarios distintos, como la programación concurrente [36], pero para un programador inexperto en el desarrollo de smart contracts, estos fallos pueden pillarle por sorpresa.

Es interesante anotar que muchas de las soluciones a los fallos estudiados requieren crear variables que se almacenan de forma permanente en storage. Esto va en contra de una de las lecciones aprendidas en la parte 1 de este documento: las aplicaciones descentralizadas deben almacenar en storage el mínimo de información posible.

Actualmente Solidity es demasiado complicado como para que un desarrollador pueda diseñar smart contracts medianamente complejos sin exponerlos a contener algunos de los fallos vistos antes o hacerlos demasiado costosos. Aunque Solidity sea un lenguaje de alto nivel, no es capaz de facilitar el desarrollo lo suficiente. Igual en futuras versiones del lenguaje se puede ver una evolución hacia un paradigma de programación funcional, que simplificará en gran medida la complejidad de los smart contracts y los haría más claros y comprensibles.

A la vista de los fallos observados durante este apartado, puede afirmarse que una aplicación como Galactic Chain es inviable como aplicación descentralizada en Ethereum. Además, las soluciones que se han tenido que diseñar en Galactic Chain para evitar los problemas presentados en este apartado hacen que la aplicación sea demasiado costosa y lenta como para ser viable.

Como conclusión, en el apartado 1 se determinó que los smart contracts son muy útiles en algunos procedimientos como intermediarios. Esto se debe a que solo hay que confiar en que el código que los regula se comporte de forma correcta. Al final de esta parte se puede afirmar que confiar en el código de un smart contract es extremadamente arriesgado. Incluso estudiando su código escrito en Solidity, es difícil determinar con total certeza que no contiene fallos. Una buena práctica, antes de utilizar una aplicación descentralizada, es obtener su código escrito en Solidity y determinar si se comporta como se espera.

Como intermediarios en procesos que pueden involucrar grandes sumas de dinero, es crucial que los smart contracts funcionen correctamente. Debido a la dificultad de determinar la corrección del código de un smart contract por un humano, actualmente se encuentra en auge el desarrollo de verificadores y analizadores de smart contracts. [33,34,35,49,50,51,52,53]

Conclusiones y trabajo futuro

A partir de las respuestas obtenidas a lo largo de este documento, se puede concluir que Ethereum es una tecnología demasiado joven. Las aplicaciones descentralizadas que en él se pueden realizar deben servir a un propósito muy concreto, como se observó en la parte 1, y deben mantenerse simples, como se observó en las partes 2 y 3.

El hecho de que Ethereum permita el desarrollo de smart contracts y que su lenguaje sea Turing completo no significa que cualquier programa tenga sentido dentro de Ethereum, como se observó en la parte 2. Las aplicaciones descentralizadas, por otro lado, pueden ser de gran utilidad como apoyo a otras aplicaciones.

En conclusión, si lo que se espera de Ethereum es que sea un blockchain, capaz de albergar aplicaciones descentralizadas, con un propósito muy definido, el de servir de intermediarias, y una complejidad muy pequeña, entonces actualmente esta capacitado para cubrir esas expectativas.

Es muy importante resaltar que Solidity es un lenguaje que, a pesar de ser de alto nivel, permite realizar operaciones de muy bajo nivel. Aunque esto permite a los desarrolladores utilizar recursos que hacen sus smart contracts más eficientes en lo que a gas respecta, supone un gran riesgo debido a lo sencillo que es introducir errores sin darse cuenta. La Fundación Ethereum es consciente de esto y cada vez impulsa más el desarrollo de lenguajes de alto nivel más restrictivos y que hagan más difícil introducir fallos en el código. Un ejemplo es Vyper [23], ya mencionado anteriormente, que incluye un fuerte sistema de tipos. Además, Vyper esta diseñado con la intención de que sea sencillo computar una cota superior precisa del gas que consume una llamada a una función.

Una de las direcciones en las que parece que está avanzando la programación de smart contract es hacia lenguajes más simples y formalmente verificables. Debido a la importancia que tiene que un smart contract funcione correctamente es un gran incentivo poder probar matemáticamente que el código de un smart contract hace lo que debe. Otras propuestas de blockchain diseñado con un lenguaje de esas características en mente son Tezos y Cardano [43].

Por otro lado son de gran interés las técnicas que se están desarrollando para hacer del blockchain una tecnología más escalable ya que esto afectará de forma directa a la capacidad de cómputo de Ethereum. Un ejemplo de estas técnicas es el paso de *proof of work* a *proof of stake*, una nueva forma de obtener consenso en un blockchain más escalable que *proof of work*. Entre los blockchains que utilizan un tipo de proof of stake se encuentran Tezos (apéndice B) y Cardano, que presenta una propuesta de *proof of stake* muy prometedora [43], aunque todavía en una fase muy preliminar.

Además, el uso de *sidechains* también parece ser prometedor tanto para aumentar la escalabilidad de un blockchain como para aumentar su capacidad de cómputo. Los *sidechains*

son bifurcaciones de un blockchain que sirven como apoyo a la rama principal y que pueden realizar cómputos de manera rápida. ¿Puede que nos acerquemos a un futuro en el que la capacidad de cómputo de un blockcchain no sea tan distinta de la de un servidor?

Como trabajo futuro es importante tener en cuenta que la tecnología blockchain es muy nueva y avanza a gran velocidad. Por ello, dado que este TFG sirve como un análisis del estado actual del blockchain de Ethereum, una de las posibilidades es realizar un estudio que determine la dirección en la que Ethereum y la tecnología blockchain pueden evolucionar y cómo va a afectar esto al uso de un blockchain como máquina de cómputo.

Otra forma de continuar con este TFG en un futuro sería extenderlo con un estudio sobre la verificación formal y el análisis de smart contracts. Como se ha podido observar en el apartado 5 no son pocos los tipos de vulnerabilidades que pueden sufrir los smart contracts y algunos de ellos son difíciles de detectar. Por este motivo esta en auge el desarrollo de técnicas que permiten analizar y verificar el código de los smart contracts, de hecho ya existen varios verificadores que pueden utilizarse. Algunos de ellos son TeEther [32], Zeus [34], Securify [35] y otras que pueden encontrarse en la bibliografía [33,34,35,49,50,51,52,53].

Conclusions and Future Work

From the answers obtained during this end-of-degree project, we can conclude that Ethereum technology is not mature. The decentralized applications that can be developed in it have to serve a really specific purpose, as we observed in the first part, and they must be simple, as we observed in parts two and three.

The fact that Ethereum allows the development of smart contracts, and that its language is Turing complete, does not mean that any software makes sense inside Ethereum, like we noticed in the second part. The decentralized applications, on the other hand, can be very useful as a support for other applications.

To sum up, if what it is expected of Ethereum is to be a Blockchain, capable of hosting decentralized applications, with a very specific purpose to serve as intermediaries and a really low complexity, then nowadays it is qualified to cover those expectations.

It's worth to say that Solidity, despite being a high-level language, allows performing very low-level operations. Although this allows the developers to use resources that make their smart contracts more efficient in terms of gas, it is a error-prone because it is extremely easy to make mistakes without noticing then. Ethereum is aware of this and it is increasingly prometing the development of high-level languages, more restrictive and that makes harder to introduce mistakes in the code. An example is Vyper [23], previously mentioned, which includes a strong type system. Moreover, Vyper is designed with the intention of being simple to compute a precise, upper bound of the gas that consumes a call to a function.

One of the trends of the smart contract programming community is towards simpler and formally verifiable languages. Due to the importance of making the smart contracts correct, it is a great incentive to be able to mathematically prove that the code of a smart contract does what it has to do. A blockchain designed with a language of those characteristics on mind is Tezos and Cardano.

On the other hand, there are very interesting the techniques that are being developed to transform blockchain into a more scalable technology, as this will directly improve the computing capacity of Ethereum. An example of these techniques is the transition from proof of work to proof of stake, a new, more scalable way to obtain consensus in a blockchain, Tezos and Cardano, that represents a proposal of proof of stake really promising [43].

Moreover, the use of sidechains seems very promising too, not only to increase the scalability of a blockchain, but to increase its computing capacity as well. The sidechains are bifurcations of a blockchain that are used to support the main branch and that can make computational steps quickly. Are we approaching a future in which the computing capacity of a blockchain won't be so different to a server's capacity?

When thinking about future work it is important to take into account the high speed in which blockchain technology is evolving because of its. Because of that, since this

end-of-degree project is meant to be an analysis of the Ethereum blockchain current state, it makes sense to extend this work making a study that points out the direction in which Ethereum and blockchain technology can evolve and how it will that affect to the use of blockchains as computing machines.

Another way in which this project could be continued is by studying about formal verification and analysis of smart contracts. As it is shown in section 5, there are many ways in which smart contracts can be vulnerable and some of those vulnerabilities are difficult to find. This is the reason why the development of techniques that allow us to analyse and to verify smart contracts code is booming. In fact, currently there are a few smart contracts verifiers that can be used as TeEther [32], Zeus [34] Securify [35] and [33,34,35,49,50,51,52,53].

Apéndice A: Proyectos reales sobre Ethereum

Dos propuestas prometedoras construidas sobre el blockchain de Ethereum

En la actualidad, tras la aparición de blockchains como Ethereum, que soportan smart contracts, las aplicaciones descentralizadas están en auge. En [18] pueden verse una lista con información de mas de 2000 aplicaciones descentralizadas que están en desarrollo o ya están operativas. Debido a este auge muchas empresas presentan proyectos de aplicaciones descentralizadas (Dapps) los cuales aún distan mucho de ser aplicaciones reales. Algunas start ups tienen una propuesta de Dapp no financiera pero carecen de información técnica al respecto por ejemplo:

-*BITNATION*: esta start up propone la creación de un gobierno desarrollado en blockchain. El gobierno lo formarían una serie de smart contracts, en ellos se guardarían datos como por ejemplo, la constitución, la identidad de sus 'habitantes'.[44]

-*Arcade City:* propone el uso de blockchain para gestionar una aplicaciones de *ride share*.[45]

-*PeerNova:* integridad y seguridad de datos en el mundo financiero.[46]

-*Follow my vote:* actualmente está desarrollando un servicio online implementado sobre blockchain que permite aportar transparencia y seguridad a los procesos electorales.[47]

En [20] se pueden encontrar 30 areas en las que se están desarrollando Dapps no financieras, entre ellas las ya mencionadas. A continuación se va a ver en detalle la propuesta de dos Dapps no financieras que actualmente sí que poseen una propuesta real, Verity y FOAM.

1.Verity, una plataforma basada en blockchain para la

1.1.obtención de datos en tiempo real

Verity presenta una API descentralizada para conseguir información fiable en tiempo real. En esta API cualquiera puede obtener dinero por compartir aquello que ve y experimenta, presentando una alternativa fiable y barata para obtener información. [15].

Esta API utiliza el principio 'wisdom of the crowd' para sacar la verdad de la información que se le proporciona.

Actualmente las APIs que proveen datos están bajo el control de una única entidad, la cual puede manipular a su antojo la información que posee. Verity sin embargo es una API descentralizada y por lo tanto más fiable. Para controlar esta red descentralizada de datos Verity utiliza smart contracts.

Para conseguir usuarios proveedores de datos, la API recompensará económicamente a aquellos proveedores que hayan compartido su información con el sistema si la información concuerda con el veredicto final producido mediante un algortime de consenso.

1.2.Arquitectura del sistema

Verity busca proveer información en **tiempo real** que quede libre de poder ser comprometida, por lo que necesita utilizar una **red descentralizada**. Proveer información en tiempo real hace imposible que todo el sistema de Verity se encuentre en un blockchain, pues el sistema necesita realizar millones de transacciones por segundo y esto no es posible en blockchain. Por otro lado Verity almacena su información en una red descentralizada la cual debe ser administrada correctamente evitando que una única entidad controle el sistema, es por ello por lo que Verity utiliza blockchain.

Estas dos características, tiempo real y autorregulación, hacen que en Verity se dé una cooperación entre una red descentralizada que realice operaciones de manera rápida y un blockchain para gestionarla.

1.3.Nodos de la red:

Nodos de validación: son los responsables de analizar los datos, construir un consenso y entregar la información a los desarrolladores. Estos nodos contribuyen a la gobernanza del sistema y la importancia de su voto depende de su reputación.

Nodos proveedores de datos: tienen una comunicación unidireccional con los nodos de validación, pues les envían los datos.

Mercados: Para obtener información de Verity un desarrollador debe definir qué clase de información quiere, qué reglas se deben cumplir para que la información sea válida y cuánto esta dispuesto a pagar por la información. A este proceso se le llama creación de un mercado. Un mercado está formado por uno o varios *data feeds que* son el conjunto de campos que representan los datos que debe administrar el proveedor o decisiones que este debe tomar.

Reglas de consenso: El desarrollador, a la hora de crear un mercado, debe definir aquellos requisitos que se deben cumplir para que esté dispuesto a pagar por la información. Un ejemplo de estos requisitos es por ejemplo el mínimo y máximo número de gente que debe rellenar los *data feeds.*

Staking: Para controlar que la calidad de los proveedores de datos sea alta, Verity utiliza un sistema parecido a una *proof of stake.* Los proveedores de datos mantienen una reputación que les da acceso a determinados mercados, esta reputación depende de si sus posteriores contribuciones a mercados han estado del lado del veredicto final o no.

Smart contracts: Son responsables de la seguridad y justicia de Verity. Son usados para llevar el *staking*, mantener los fondos, el sistema de reputación y la gobernanza. Un desarrollador,

cuando crea un mercado(que es un smart contract), guarda en él la recompensa monetaria que dará por la información y un hash de las reglas de consenso. En el smart contract también se almacenarán los nodos validación de ese mercado y los proveedores de datos.

Selección de los nodos de validación: Los nodos de validación son seleccionados de manera aleatoria y añadidos al smart contract, no son conocidos de ahí en adelante y por ello ni los desarrolladores ni los proveedores de datos pueden influir en ellos. La selección aleatoria se hace mediante una función de blockchain basada en el hash del último bloque.

2.FOAM, el mapa del mundo dirigido por consenso

En la actualidad el uso de mapas digitales está cada vez más extendido. No solo nos permiten obtener información acerca de la situación geográfica de determinado lugar. Nos guían a la hora de movernos por nuestro entorno, nos proporcionan información de aquello que nos rodea e incluso nos recomiendan servicios como restaurantes, tiendas, etc.

Actualmente el referente en mapas digitales es Google Maps y se encuentra seguida de una serie de empresas privadas. Las iniciativas de crear una aplicación de mapas de código abierto no han alcanzado hasta el momento la competitividad necesaria debido a la falta de fondos.

FOAM [16] pretende proporcionar a las aplicaciones la opción de poder incluir mapas sin tener que comprometer la descentralización y la seguridad de la aplicación. Desde FOAM opinan que los usuarios deberían ser los poseedores de la información sobre su localización personal y por ello deben poder decidir con quién la comparten y cuándo.

Dentro del funcionamiento de FOAM se destacan tres elementos: **Crypto spatial coordinate standard**, **spatial Index and Visualizer web app** and **proof of location**.

2.1.Crypto spatial coordinate (CSC) standard

FOAM utiliza un mecanismo para relacionar una dirección en el blockchain con una dirección geográfica. De esta forma todo smart contract puede hacer referencia a determinada dirección en el mapa y, por lo tanto, las transacciones entre smart contracts pueden tomar un plano espacial. Las direcciones físicas se obtienen mediante el estándar geohash [19], que es de dominio publico y permite codificar localizaciones geográficas en cadenas de letras y dígitos [17]. Otra de las ventajas de utilizar geohashes para representar localizaciones es que sus hashes son intrínsecamente jerárquicos, por lo tanto localizaciones que tienen una relación espacial darán lugar a hashes relacionados.

El protocolo utilizado para crear una dirección CSC toma como entrada una dirección geohash y una dirección de Ethereum. A partir de estas dos se creará una dirección que representa la unión de ambas entradas. Cuanto mayor es la dirección de salida, más específica es

la localización que representa. El protocolo permite mappear la dirección de un contrato y el geohash de la dirección geográfica a partir del CSC que forman.

2.2.Spatial Index and Visualizer(SIV)

FOAM cuenta con un explorador visual del blockchain de propósito general. El SIV puede ser utilizado como una interfaz *frontend* para cualquier Dapp que necesite visualizar smart contracts en un mapa [16].

Las CSCs permiten que los smart contracts tengan una representación espacial. Es por ello por lo que todo smart contract que formen parte de una CSC puede representarse en un mapa. SIV es una aplicación web diseñada para que los usuarios puedan interactuar con los smart contracts que utilicen CSCs y, por otro lado, para que otras Dapps que se construyen sobre el protocolo de FOAM puedan utilizarlo.

2.3.Proof of location

El objetivo de este elemento es presentar una alternativa descentralizada al GPS que sea segura, precisa y resistente a la censura. Una de las principales ventajas de los mapas de Google con respecto al resto de mapas es la gran cantidad de puntos de interés (POI) de los que mantienen información. Para que FOAM represente una alternativa competitiva debe encontrar una manera de incluir puntos de interés que sea fiable. Para mantener esta información, FOAM utiliza los TCRs (token curated registries). Los TCRs son un modelo cryptoeconómico mediante el cual determinados usuarios pueden obtener incentivos económicos por mantener la veracidad de una lista de contenidos. De esta manera en FOAM aparece un nuevo rol, el de los cartógrafos, siendo estos los usuarios que se encargan de que los TCRs sean listas con información cierta.

Dentro de los TCRs hay tres actores: los consumidores, que utilizan la lista de contenidos, los candidatos; que quieren añadirse a la lista de contenidos como punto de interés; y los cartógrafos, que se encargan de mantener la validez de la lista de contenidos.

El funcionamiento de un TCR sería el siguiente:

1. Cuando un candidato quiere añadir un POI al TCR realiza un depósito de tokens de FOAM en el TCR con el correspondiente CSC.
2. El candidato tendrá que pasar una serie de desafíos para convencer a los cartógrafos de que el POI que propone debe estar en el TCR. Si, tras el periodo de prueba, ningún cartógrafo se opone a la inclusión de tal POI este será añadido al TCR y el deposito queda asegurado en el TCR y asociado a ese POI.
3. Si durante la fase de prueba algún cartógrafo se opone a la inclusión de ese POI, este puede proponer un desafío al candidato siempre y cuando envíe la misma cantidad de tokens que el candidato al TCR. Esto iniciará un periodo de votación entre los cartógrafos.
4. Tras el periodo de votación, si el cartógrafo que se oponía tiene éxito, este y los cartógrafos que votaron a su favor se distribuirán el deposito del candidato.

5. Si el reto del cartógrafo no tuvo éxito, entonces el deposito que realizó se lo repartirán entre el candidato cuyo POI fue retado y los cartógrafos que votaron a su favor y por lo tanto ganaron la votación.

La interacción con el TCR se realiza a través del SIV. En Ethereum las transacciones tardan 15 segundos y lo que se espera de un mapa interactivo, como google maps, es que proporcione la información que se necesita de forma rápida. Por este motivo, FOAM no puede ser una aplicación totalmente descentralizada.

Apéndice B: ATOMICCALL

una nueva operación en la EVM

1.Introducción

Del mismo modo que la operación STATICCALL se añadió a la EVM para representar interacciones que no iban a alterar el estado de esta, en este apéndice veremos cómo se puede modificar la definición formal de Ethereum [7] para introducir una nueva operación que llamaremos ATOMICCALL. Realizando los cambios necesarios para permitir la ejecución de esta operación.

2.Funcionamiento de ATOMICCALL

ATOMICCALL es una operación que se usará para interactuar con la EVM del mismo modo que lo hace la operación CALL. A diferencia de STATICCALL, esta operación sí puede modificar el estado de la EVM. La que diferencia a ATOMICCALL del resto de operaciones que realizan una llamada a otro contrato en la EVM es que las acciones que se quieren ejecutar sobre una cuenta (código de un smart contract o transferencias de ether), al indicarse a través de una ATOMICCALL, se ejecutarán si y sólo si el estado del receptor al comienzo de la ejecución de la transacción es el mismo al estado que tenía cuando se emitió la transacción, evitando el problema del orden expuesto en el apartado 5.3. Por lo tanto una definición informal sería:

$ATOMICCALL(a, ops[], ...) \equiv exe(a, ops[], ..)$ *sí* $\neg M(E(a))$
$ATOMICCALL(a, ops[], ...) \equiv \varnothing$ *en otro caso*

Donde *exe(a,ops[],...)* representa la ejecución de las operaciones *ops[]* sobre la cuenta *a*. *E(a)* representa el estado de *a*. *M(E(a))* por lo tanto devuelve cierto si el estado de *a* se ha modificado desde que se envió la llamada a la EVM hasta antes de ejecutar *ops[]*.

3.¿Por qué la EVM necesita una operación así?

La EVM es el entorno en el cual se almacena toda la información sobre las cuentas de Ethereum y es donde se realizan las operaciones que llevan a estas cuentas de un estado a otro. El estado de la EVM cambia cuando cambia el estado de alguna de las cuentas que contiene.

El diseño de la EVM asegura que todas las transacciones se ejecutan en exclusión mutua. Si esto es así parece innecesario necesitar una operación como ATOMICCALL ya que si las transacciones se ejecutan en exclusión mutua el estado de la EVM no puede cambiar en el tiempo que transcurre entre la emisión de la transacción y su ejecución. El fallo de este razonamiento está en olvidar el hecho de que la EVM se encuentra en un blockchain y por lo tanto un gran número de transacciones pueden ser emitidas a la vez. Es cierto que, aunque se emitan a la vez, se ejecutarán en exclusión mutua, pero el orden en el que se ejecuten sólo podrá conocerse una vez se hayan minado los bloques que contienen esas transacciones.

Esto podría provocar la siguiente situación. Sean t_a y t_b dos transacciones que se han emitido a la vez. La transacción t_a va a realizar una operación sobre el smart contract *S* que se encontraba en el estado S_e cuando t_a y t_b se emitieron. Por otro lado t_b va a modificar el estado de S pasando a ser su estado S_e'. Podría suceder que en el nuevo bloque que se ha minado, y en el que ya se han ejecutado t_a y t_b en la EVM, primero se ha ejecutado t_b y después t_a. Esto significa que t_a, que estaba emitida con la intención de realizar una serie de operaciones sobre el estado S_e, ha realizado las operaciones sobre el estado S_e' que ha dejado la ejecución de la transacción t_b.

Podría argumentarse que estos problemas deberían tenerlos en cuenta quienes programan smart contracts para la EVM y por lo tanto no es necesario modificar la EVM. Si bien es una solución válida esta provoca un aumento en la complejidad de los smart contracts y de las llamadas a sus funciones.

Es por esto por lo que la solución que menos repercusión tendría sobre la complejidad del diseño de smart contracts parece ser la de modificar la EVM para incluir la operación ATOMICCALL.

4.Modificaciones en la definición formal de la EVM para soportar ATOMICCALL

En primer lugar se introducirán algunas de las definiciones que el *yellow paper* [7] de Ethereum presenta y que se utilizará durante este apartado.

El estado global de la EVM se representa mediante la variable σ y dada una cuenta 'a' esta variable contiene la siguiente información sobre dicha cuenta:

- ***nonce:*** *número de transacciones enviadas por dicha cuenta.* $\sigma[a]_n$
- ***balance:*** *fondos de la cuenta.* $\sigma[a]_b$
- ***storageRoot:*** *el hash de la raíz del Merkle Patricia tree que codifica el estado de cada una de las variables que la cuenta mantiene en storage.* $\sigma[a]_s$
- ***codeHash:*** *hash del código que almacena esta cuenta.* $\sigma[a]_c$

La información que más importancia tiene para conseguir implementar la operación ATOMICCALL es el ***storageRoot*** ya que el estado de un smart contract cambia cuando sus variables en *storage* lo hacen. La EVM debe poder comprobar que el valor $\sigma[a]_s$ para el que se envió la operación ATOMICCALL no ha cambiado cuando empiece a ejecutarse el código de *a* que se indica la operación.

El *yellow paper* de Ethereum define la ejecución de una transacción T como la transformación de σ a σ'. Esta tiene una serie pasos de intermedios.

Antes de ejecutar la transacción se crea, a partir de σ, el checkpoint σ_0 que es igual a σ pero el **balance** y el **nonce** del emisor de la transacción se han modificado para cobrar el gas que se usará en la ejecución de la transacción y registrar el envío de esta.

Si la transacción contiene algún tipo de llamada como operación, la ejecución se representa de la siguiente manera:

$^{\ominus}(\sigma_0,s,o,r,c,g,p,v,v',d,e,w)$

Donde *s* es quien realiza la llamada, *o* es quien creó la transacción, *r* es la cuenta que recibe el ether que se manda, *c* es la cuenta cuyo código se va a ejecutar, *g* es la cantidad de gas disponible, *v* es la cantidad de ether que se envía, *p* es el precio del gas en este momento, *d* es un array con el input del código que se va a ejecutar, *e* es la profundidad en la pila en la que se encuentra esta llamada y por último *w* es el permiso para modificar el estado.

El entorno de ejecución de las operaciones de la EVM cuenta con la información que contiene la tupla I, que es la siguiente:

I_a, address de la cuenta a la cual pertenece el código que está ejecutándose.

I_o, address de la cuenta que emitió la transacción que ha originado esta ejecución.

I_p, precio del gas en la transacción que originó esta ejecución.

I_d, el array de bytes que contiene la entrada para esta ejecución.

I_s, el address de la cuenta que ha causado que el código se esté ejecutando.

I_v, valor en *Wei* enviado a esta cuenta.

I_b, array de bytes con que representa el código que va a ejecutarse.

I_H, cabecera del bloque actual (el que se está minando).

I_e, la profundidad de la llamada que se está ejecutando.

I_w, permiso para modificar el estado.

Este entorno no es suficiente para poder ejecutar la llamada ATOMICCALL. Durante la ejecución, aunque sí se puede saber el estado en el que se encuentra el storage de la cuenta $\sigma[I_a]_s$, no se puede comparar con el storage que tenía I_a cuando se emitió la llamada.

La solución que se propone para obtener la información que falta comienza por razonar que el *storage* que tenía I_a cuando se emitió la transacción es el storage que tiene en el bloque anterior al que se está minando. La información que almacena un bloque se muestra en la siguiente imagen obtenida de [7].

parentHash: The Keccak 256-bit hash of the parent block's header, in its entirety; formally H_p.
ommersHash: The Keccak 256-bit hash of the ommers list portion of this block; formally H_o.
beneficiary: The 160-bit address to which all fees collected from the successful mining of this block be transferred; formally H_c.
stateRoot: The Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied; formally H_r.
transactionsRoot: The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block; formally H_t.
receiptsRoot: The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block; formally H_e.
logsBloom: The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list; formally H_b.
difficulty: A scalar value corresponding to the difficulty level of this block. This can be calculated from the previous block's difficulty level and the timestamp; formally H_d.
number: A scalar value equal to the number of ancestor blocks. The genesis block has a number of zero; formally H_i.
gasLimit: A scalar value equal to the current limit of gas expenditure per block; formally H_l.
gasUsed: A scalar value equal to the total gas used in transactions in this block; formally H_g.
timestamp: A scalar value equal to the reasonable output of Unix's time() at this block's inception; formally H_s.
extraData: An arbitrary byte array containing data relevant to this block. This must be 32 bytes or fewer; formally H_x.
mixHash: A 256-bit hash which, combined with the nonce, proves that a sufficient amount of computation has been carried out on this block; formally H_m.

Lo que se propone aquí es añadir dentro de un bloque la siguiente información:

parentState: una estructura que contiene el **storageRoot** de todas las cuentas en el bloque del cual el actual es hijo. Se representa como $\mathbf{H}_r$ y, dada la cuenta 'a', $\mathbf{H}_r$[a] es el **storageRoot** de dicha cuenta en el bloque anterior al actual.

De este modo se puede definir la operación ATOMICCALL de la siguiente manera:

Valor: 0xf5

Mnemonic: ATOMICCALL

Descripción:

Envía un mensaje a una cuenta cuyo código sólo se ejecutará si el storage de dicha cuenta no ha cambiado desde que la transacción que provoca esta ejecución se emitió. Es equivalente a CALL a excepción de:

Sí y solo sí $\mu_s[2] \leq \sigma[I_a]_b \wedge I_e < 1024 \wedge \mathbf{(I_H)_r[t] = \sigma[I_a]_r}$ entonces:

La función CALL se define de la siguiente forma, todas las variables están definidas en [7]:

0xf1 CALL 7 1 Message-call into an account.

$$\mathbf{i} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[3] \dots (\boldsymbol{\mu}_\mathbf{s}[3] + \boldsymbol{\mu}_\mathbf{s}[4] - 1)]$$

$$(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, I_a, I_o, t, t, & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_a]_b \wedge \\ \quad C_{\text{CALLGAS}}(\boldsymbol{\mu}), I_p, \boldsymbol{\mu}_\mathbf{s}[2], \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_e + 1) & I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, \mathbf{o}) & \text{otherwise} \end{cases}$$

$$n \equiv \min(\{\boldsymbol{\mu}_\mathbf{s}[6], |\mathbf{o}|\})$$

$$\boldsymbol{\mu}'_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[5] \dots (\boldsymbol{\mu}_\mathbf{s}[5] + n - 1)] = \mathbf{o}[0 \dots (n-1)]$$

$$\boldsymbol{\mu}'_g \equiv \boldsymbol{\mu}_g + g'$$

$$\boldsymbol{\mu}'_\mathbf{s}[0] \equiv x$$

$$A' \equiv A \Cup A^+$$

$$t \equiv \boldsymbol{\mu}_\mathbf{s}[1] \mod 2^{160}$$

where $x = 0$ if the code execution for this operation failed due to an exceptional halting $Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) = \top$ or if $\boldsymbol{\mu}_\mathbf{s}[2] > \boldsymbol{\sigma}[I_a]_b$ (not enough funds) or $I_e = 1024$ (call depth limit reached); $x = 1$ otherwise.

$$\boldsymbol{\mu}'_i \equiv M(M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_\mathbf{s}[3], \boldsymbol{\mu}_\mathbf{s}[4]), \boldsymbol{\mu}_\mathbf{s}[5], \boldsymbol{\mu}_\mathbf{s}[6])$$

Thus the operand order is: gas, to, value, in offset, in size, out offset, out size.

$$C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$$

$$C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + G_{callstipend} & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{otherwise} \end{cases}$$

$$C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} \min\{L(\boldsymbol{\mu}_g - C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})), \boldsymbol{\mu}_\mathbf{s}[0]\} & \text{if } \boldsymbol{\mu}_g \geq C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \\ \boldsymbol{\mu}_\mathbf{s}[0] & \text{otherwise} \end{cases}$$

$$C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{call} + C_{\text{XFER}}(\boldsymbol{\mu}) + C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$$

$$C_{\text{XFER}}(\boldsymbol{\mu}) \equiv \begin{cases} G_{callvalue} & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{newaccount} & \text{if } \boldsymbol{\sigma}[\boldsymbol{\mu}_\mathbf{s}[1] \mod 2^{160}] = \varnothing \\ 0 & \text{otherwise} \end{cases}$$

Una de las desventajas de esta solución es que se añade más información en los bloques del blockchain y por lo tanto ocupará más memoria. Sin embargo, de este modo, las cuentas que quieran asegurarse de que su transacción se ejecutará sobre otra cuenta que se encuentra en el momento de ejecución en el mismo estado que tenía cuando se emitió la transacción.

Referencias

[1]	Nakamoto, Satoshi, “Bitcoin: A Peer-to-Peer Electronic Cash System”. URL: https://bitcoin.org/bitcoin.pdf
[2]	“Stop worrying about how much energy bitcoin uses”.URL: https://theconversation.com/stop-worrying-about-how-much-energy-bitcoin-uses-97591
[3]	Precio actual de Bitcoin. URL: https://www.coinbase.com/price/bitcoin.
[4]	Encriptación clave pública-clave privada en detalle.URL: https://en.wikipedia.org/wiki/Public-key_cryptography
[5]	whitepaper de Ethereum. URL:http://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf

[6]	Explicación detallada de los Patricia Merkle trees. URL:https://github.com/ethereum/wiki/wiki/Patricia-Tree
[7]	Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. URL: https://ethereum.github.io/yellowpaper/paper.pdf
[8]	Página web de Namecoin. URL: https://namecoin.org/
[9]	wiki de Colored coins.URL: https://en.bitcoin.it/wiki/Colored_Coins
[10]	Explicación de Ethereum de Vitalik Buterin. URL: https://www.youtube.com/watch?v=66SaEDzlmP4
[11]	Documentación de Tezos para desarrolladores. URL: https://tezos.gitlab.io/master/index.html
[12]	Proof of work y proof of stake desde el punto de vista de un minero. URL: https://blockgeeks.com/guides/proof-of-work-vs-proof-of-stake/
[13]	Documentación de liquidity. URL: http://www.liquidity-lang.org/doc/index.html
[14]	Comparación entre el gasto de energía de Bitcoin y el gasto de algunos países. URL: https://powercompare.co.uk/bit- coin/
[15]	Whitepaper proporcionado por verity con la información técnica de su sistema. URL: http://verity.network/whitepaper.pdf
[16]	Whitepaper proporcionado por FOAM con la información técnica de su sistema. URL: https://www.foam.space/publicAssets/FOAM_Whitepaper.pdf
[17]	Información acerca de geohash. URL: https://en.wikipedia.org/wiki/Geohash
[18]	Registro aplicaciones descentralizadas. URL: https://www.stateofthedapps.com
[19]	Estándar geogash. URL: http://geohash.org
[20]	Casos de aplicaciones descentralizadas no financieras.

	URL:https://gomedici.com/30-non-financial-use-cases-of-blockchain-technology-infographic/
[21]	Documentación de Solidity. URL: https://solidity.readthedocs.io/en/v0.5.7/index.html
[22]	Remix. URL: https://remix.ethereum.org/
[23]	Documentación de Vyper. URL: https://vyper.readthedocs.io/en/v0.1.0-beta.9/index.html
[24]	Guía de instalación de Solc. URL: https://www.npmjs.com/package/solc
[25]	Read the docs de Web3.js. URL: https://web3js.readthedocs.io/en/1.0/#
[26]	Read the docs de Web3.py. URL: https://web3py.readthedocs.io/en/stable/
[27]	Documentación Web3j. URL: https://docs.web3j.io/getting_started.html
[28]	Especificación JSON-RPC. URL: https://www.jsonrpc.org/specification
[29]	Documentación de JavaScript. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript
[30]	Web de Infura. URL: https://infura.io/dashboard
[31]	Web de Metamask. URL: https://metamask.io/
[32]	Johannes Krupp, Christian Rossow, TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts. 27th USENIX Security Symposium (USENIX Security 18) URL:https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-krupp.pdf
[33]	Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, Yannis Smaragdakis, MadMax: Surviving Out-of-Gas Conditions in Ethereum. Smart Contracts. Proc. ACM Program. Lang. November 2018 URL: http://doi.acm.org/10.1145/3276486

[34]	Sukrit Kalra, Seep Goel, Mohan Dhawan, Subodh Sharma, ZEUS: Analyzing Safety of Smart Contracts.Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018 URL:http://pages.cpsc.ucalgary.ca/~joel.reardon/blockchain/readings/ndss2018_09-1_Kalra_paper.pdf
[35]	Peter Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, Martin Vechev, Securify: Practical Security Analysis of Smart Contracts. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. URL: https://arxiv.org/pdf/1806.01143.pdf
[36]	Maurice Herlihy, Blockchains from a distributed computing perspective.Commun. ACM. February 2019.URL: http://doi.acm.org/10.1145/3209623
[37]	Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, Aquinas Hobor, Making Smart Contracts Smarter. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. URL: http://doi.acm.org/10.1145/2976749.2978309
[38]	Criptodivisas y su valor actual. URL: https://coinmarketcap.com/all/views/all/
[39]	Información de Ethereum en tiempo real. URL: https://etherscan.io
[40]	Página de Bitnation, un proyecto para crear una nación gobernada por a través del blockchain de Ethereum. URL: https://bitnation.consider.it/?tab=Show%20all
[41]	Blockchain cuties, juego diseñado en Ethereum que consiste en criar mascotas virtuales. URL: https://blockchaincuties.com/?utm_source=StateOfTheDApps
[42]	Artículo sobre sidechains. URL: https://thenextweb.com/hardfork/2018/12/11/blockchain-sidechains-explained-basics/

[43]	Artículos académicos de Cardano. URL: https://www.cardano.org/en/home/ Bitnation. URL: https://tse.bitnation.co/
[44]	Arcade city. URL:https://arcade.city/roadmap
[45]	PeerNova. URL:https://peernova.com/
[46]	Follow my vote. URL: https://followmyvote.com/
[47]	Registro en línea del blockchain de Rinkeby. URL: https://rinkeby.etherscan.io/
[48] [49]	Elvira Albert, Jesús Coreas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. SAFEVM: A Safety Verifier for Ethereum Smart Contracts. In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2019 (ISSTA'19), ACM, 2019. To appear.
[50]	Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In Shuvendu Lahiri and Chao Wang, editors, 16th International Symposium on Automated Technology for Verification and Analysis, ATVA 2018. Proceedings, volume 11138 of Lecture Notes in Computer Science, pages 513-520. Springer, 2018.
[51]	Oyente: An Analysis Tool for Smart Contracts, 2018. URL: https://github.com/melonproject/oyente.
[52]	S. Amani, M. Bégel, M. Bortin, and M. Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In CPP. ACM, 66–77.
[53]	I. Grishchenko, M. Maffei, and C. Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In POST (LNCS), Vol. 10804. Springer, 243–269.
[54]	Vitalik Buterin en Deconomy, Korea, 2018. URL: https://www.youtube.com/watch?v=7WL9hr445uo&t=52s

[55]	Repositorio código Galactic Chain. URL: https://github.com/HermenegildoTFG/GalacticChain
[56]	Wikipedia de Ethereum. URL: https://en.wikipedia.org/wiki/Ethereum